NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

APPLICATION OF MULTI-FREQUENCY MODULATION (MFM) FOR HIGH-SPEED DATA COMMUNICATIONS TO A VOICE FREQUENCY CHANNEL

by

Charles P. Salsman

June 1990

Thesis Advisor: P. H. Moose

Unclassified

security classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification Unclassified			1b Restrictive Markings
2a Security Classification Authority			3 Distribution/Availability of Report
2b Declassification/Downgrading Schedule			Approved for public release; distribution is unlimited.
4 Performing Organization Report Number(s)			5 Monitoring Organization Report Number(s)
6a Name of Performing Organization Naval Postgraduate School	6b Office Symbol *(if applicable)* 32		7a Name of Monitoring Organization Naval Postgraduate School
6c Address *(city, state, and ZIP code)* Monterey, CA 93943-5000			7b Address *(city, state, and ZIP code)* Monterey, CA 93943-5000
8a Name of Funding/Sponsoring Organization	8b Office Symbol *(if applicable)*		9 Procurement Instrument Identification Number
8c Address *(city, state, and ZIP code)*			10 Source of Funding Numbers
			Program Element No / Project No / Task No / Work Unit Accession No

11 Title *(Include security classification)* APPLICATION OF MULTI-FREQUENCY MODULATION (MFM) FOR HIGH-SPEED DATA COMMUNICATIONS TO A VOICE FREQUENCY CHANNEL

12 Personal Author(s) Charles P. Salsman

13a Type of Report	13b Time Covered	14 Date of Report *(year, month, day)*	15 Page Count
Master's Thesis	From To	June 1990	105

16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms *(continue on reverse if necessary and identify by block number)*
Field	Group	Subgroup	communications, multi-frequency modulation, high-speed modems

19 Abstract *(continue on reverse if necessary and identify by block number)*

Multi-Frequency Modulation (MFM) has been developed at NPS using both differential quadrature-phase-shift-keying (DQPSK) and differential-quadrature- amplitude-modulation (DQAM) encoding formats. Previous applications of these encoding formats were on industry standard computers (PC) over a 16-20 kHz channel.

This report discusses the implementation of MFM to a voice frequency channel of 200-3400 Hz, for possible future use with high-speed modems over switched telephone networks. Research and testing for this report included the DQPSK and differential 16-quadrature-amplitude-modulation (D16-QAM) encoding formats implemented on PCs. Experimental results of the implemented MFM signal were comparable to theory with acceptable bit error rates for input signal-to-noise ratios (SNR) of 15 dB and higher.

20 Distribution/Availability of Abstract	21 Abstract Security Classification	
☒ unclassified unlimited ☐ same as report ☐ DTIC users	Unclassified	
22a Name of Responsible Individual P.H. Moose	22b Telephone *(Include Area code)* (408) 646-2838	22c Office Symbol 62Me

DD FORM 1473,84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

security classification of this page

Unclassified

Approved for public release; distribution is unlimited.

Application of Multi-Frequency
Modulation (MFM) for High-Speed
Data Communications to a Voice
Frequency Channel

by

Charles P. Salsman
Lieutenant Commander, United States Navy
B.S.I.E., University of Tennessee, 1977

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1990

Author: Charles P. Salsman

Approved by: P.H. Moose, Thesis Advisor

J.H. Miller, Second Reader

John P. Powers, Chairman,
Department of Electrical and Computer Engineering

ABSTRACT

Multi-Frequency Modulation (MFM) has been developed at NPS using both differential quadrature-phase-shift-keying (DQPSK) and differential-quadrature-amplitude-modulation (DQAM) encoding formats. Previous applications of these encoding formats were on industry standard computers (PC) over a 16-20 kHz channel.

This report discusses the implementation of MFM to a voice frequency channel of 200-3400 Hz, for possible future use with high-speed modems over switched telephone networks. Research and testing for this report included the DQPSK and differential 16-quadrature-amplitude-modulation (D16-QAM) encoding formats implemented on PCs. Experimental results of the implemented MFM signal were comparable to theory with acceptable bit error rates for input signal-to-noise ratios (SNR) of 15 dB and higher.

THESIS DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

As technological improvements to communication equipment and systems have been made, they have become more digital and less analog. This is primarily because the advantages of digital communication far outweigh their disadvantages. Some of these advantages include:

- Digital circuits are more reliable and can be produced at lower cost than can analog circuits.
- Digital equipment is more flexible than analog equipment.
- Digital signals can be regenerated much easier than analog signals. Although digital transmissions are degraded by electrical noise and other interferences, the original transmitted digital pulse can be regenerated using digital signal processing.
- Digital circuits are less subject to distortion and interference than analog circuits. Since binary digital circuits are always in one of two states (fully-on or fully-off), it takes a large disturbance to incorrectly change the state from one to the other.

With the digitizing of communications, and the increased use of industry standard personal computers (PC) for information exchange, the need has arisen for a signal modulation scheme that can be easily adapted to a variety of communication mediums, and that can emulate most existing modulation formats and generate new formats. Multi-Frequency Modulation (MFM) is a modulation technique that suits these needs well. It utilizes the hardware and software of the host computer to modulate and multiplex, demultiplex and demodulate the signal, thereby eliminating the requirement for analog equipment to perform these functions. MFM allows flexibility and utilizes existing PC hardware with minor upgrades by the addition of expansion boards.

The focus of this thesis is the application of MFM to a voice frequency channel over which high-speed (9600 bits per second (bps) or higher) modems will communicate. Presently no standard high-speed (or low-speed) modems use this modulation technique for transmission of data.

Chapter II gives functional descriptions of high-speed modems and of the voice-frequency switched telephone lines. The focus of Chapter III is the implementation of the voice-frequency MFM system, including hardware and software. Two differential Gray-encoding techniques were implemented with software on PCs for this thesis. The encoding formats are Differential Quadrature-Phase-Shift-Keying (DQPSK), and Differential 16-Quadrature-Amplitude-Modulation (D16-QAM). A performance evaluation was conducted on each technique and the results are discussed and analyzed in Chapter IV. Chapter V contains conclusions and recommendations.

B. THEORY OF MULTI-FREQUENCY MODULATION

The following sections provide an overview of the theory of MFM. The reader is referred to Refs. 1 and 2 for a more detailed description.

1. MFM Signal Packet

The basic structure of MFM is time and frequency slots. The MFM signals are actually sets of multiple tones which are grouped into "packets". These packets are arbitrarily located in the frequency spectrum and in time. They consist of one or more bauds. The following terms are used in the description of MFM:

T: Packet length in seconds

ΔT: Baud length in seconds

L: Number of bauds per packet

$\Delta f = 1/\Delta T$: Frequency spacing between MFM tones

k_x: Baud length in number of samples

Δt: Time between samples in seconds

$f_x = 1/\Delta t$: Sampling frequency in Hz for D/A and A/D conversion

K: Number of MFM tones

ϕ_{lk}: Phase of the k^{th} tone in the l^{th} baud

A_{lk}: Amplitude of the k^{th} tone in the l^{th} baud

An MFM signal packet is shown in Figure 1. Each packet is comprised of L bauds and K tones. The information to be transmitted is independently amplitude and/or phase modulated onto the K tones. An orthogonal set is formed by these LK subsignals. In DQPSK, a single bit of information is carried by both the in-phase and quadrature components of each tone. In D16-QAM, two bits of information are carried by both the in-phase and quadrature components of each tone. The multiple tones that are present in an MFM packet are superimposed, i.e., occur simultaneously, during a subinterval of the packet called a baud [Ref. 1].

The sampling frequency is $f_x = k_x \Delta f$, since $\Delta t = \Delta T / k_x$. The Nyquist sampling theorem requires that f_x be greater than twice the highest frequency contained in the signal frequency spectrum. Conversely, Nyquist requires that the highest frequency used in the signal be less than $f_x/2$. Consequently, an MFM

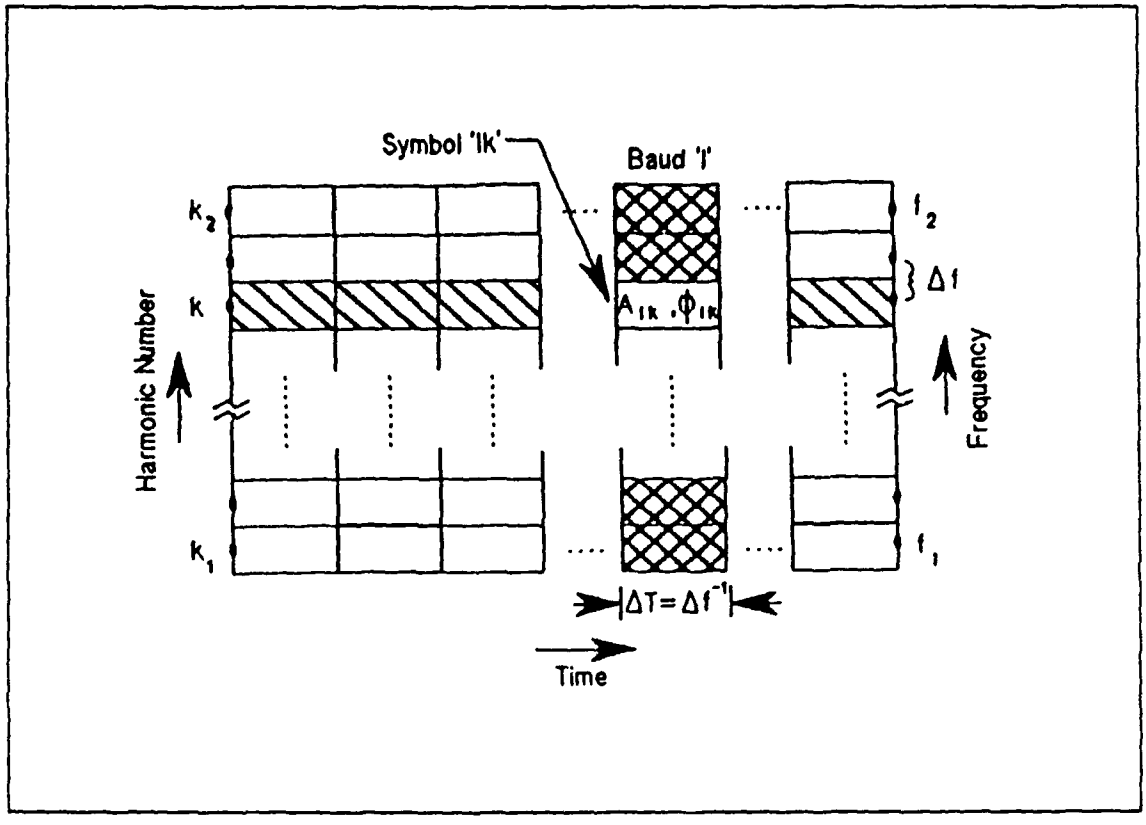


Figure 1. MFM signal packet: (after Ref. 1: p. 3).

baud is limited to a maximum of $k_x/2 - 2$ harmonic tones spaced at Δf Hz intervals from Δf Hz up to $f_x/2 - \Delta f$ Hz.

An MFM signal packet includes harmonics between $k_1 = f_1/\Delta f$ and $k_2 = f_2/\Delta f$. The values of k_1 and k_2 are chosen to generate a signal in a given passband anywhere between Δf and $f_x/2 - \Delta f$. The voice frequency band between 200 Hz and 3400 Hz is the subject of this thesis. Harmonics outside the desired passband are assigned zero amplitude. The number of tones sent in a baud is $K = k_2 - k_1 + 1$. The signal passband is $W = K \times \Delta f$. The time bandwidth product of the whole signal packet, $TW = L\Delta T \times K\Delta f$, is equal to LK, the total

number of symbols that can be sent in one packet. The MFM parameters selected for the voice frequency channel used in this thesis are shown in Table 1 below.

Table 1. DESIGN PARAMETERS FOR A 1/2.5 SECOND MFM SIGNAL PACKET IN A 200-3400 HZ PASSBAND

Baud length(sec)	ΔT	1/40	1/20	1/10	1/5	1/2.5
No. of bauds	L	16	8	4	2	1
Tone spacing	Δf	40	20	10	5	2.5
Lowest harmonic	k_1	5	10	20	40	80
Lowest tone freq.	f_1	200	200	200	200	200
Highest harmonic	k_2	85	170	340	680	1360
Highest tone freq	f_2	3400	3400	3400	3400	3400
Samples per baud	k_x	256	512	1024	2048	4096
Sampling freq	f_x	10240	10240	10240	10240	10240
No. of tones	$k_2 - k_1$	80	160	320	640	1280

The mathematical representation of the analog signal packet during the l^{th} baud is

$$x_l(t) = \sum_{k=1}^{k_x/2-1} A_{lk} \cos(2\pi k \Delta f t + \phi_{lk}), \quad (l-1)\Delta T \le t \le l\Delta T, \tag{1}$$

and the corresponding sampled discrete time signal of length k_x samples is [Ref. 2: pp. 2-3]

$$x_l(n) = \sum_{k=1}^{k_x/2-1} A_{lk} \cos\left(\frac{2\pi kn}{k_x} + \phi_{lk} \right), \quad 0 \le n \le k_x - 1. \tag{2}$$

2. MFM Generation and Demodulation

The generation and demodulation of MFM is accomplished through application of the properties of the Fast Fourier Transform (FFT) algorithm. To generate the signal, software in the transmit PC is used to load the amplitudes and phases of the MFM signal into the first half of a complex-valued array of length k_x for all tones between k_1 and k_2. The second half of the complex array is then loaded with the complex conjugate images of the values that were loaded into the first half of the array. The Inverse FFT (IFFT) is then computed to create a real signal sequence $x_l(n)$ containing k_x values as given in (2). This process is repeated until all bauds in the MFM signal have been processed. The resulting MFM signal packet consists of k_xL real values. The modulated transmitted signal $x_l(t)$ is obtained by sampling $x_l(n)$ through a digital-to-analog (D/A) converter at f_x samples per second.

Demodulation of MFM is simply the inverse of the process used to generate it. The received analog signal $y_l(t)$ is processed back into a digital signal format in the receiving PC with an analog-to-digital (A/D) converter sampling the signal at a rate of f_x times per second. The resulting k_x real values are loaded

into a k_x-point complex array, while the imaginary parts are set to zero. The FFT of the array is computed yielding, in the absence of noise, the values of A_{lk} and ϕ_{lk} which were used in the generation of the transmitted signal.

II. SYSTEM DEVELOPMENT

This chapter provides an overview of high-speed modems, the PSTN and private telephone network, and the differential encoding schemes used for this thesis. The actual development of the MFM system is not covered in this thesis because it is covered sufficiently in other references. The reader is directed to Gantenbein [Ref. 2] and Basil [Ref. 3] for a detailed description of the MFM system development.

A. FUNCTIONAL DESCRIPTION OF HIGH-SPEED MODEMS

The definition of a high-speed modem varies from publication to publication, but for the purpose of this thesis, high-speed modems in the voice frequency band are considered those with a data signaling rate of 9600 bps or higher. Specifically, the focus of this section is on full- duplex modems with a data signaling rate of 9600 bps. The international standard set by the International Telegraph and Telephone Consultative Committee (CCITT) for the 9600 bps modem, referred to as the V.32 modem, is the standard for most U.S.-manufactured full-duplex 9600 bps modems. This standard is the basis for the discussion in the following paragraphs. The purpose of including this section on modems is to give the reader a broad overview of the workings of high-speed modems. Presently, MFM cannot be applied to modems, because the modulation schemes currently used with modems are coded internally to the device and the actual modulation hardware is proprietary. If MFM is to be applied to modems, a redesign of existing modems or a new design from the ground up is necessary. This section along with the

references should provide a background for accomplishing this for future application of MFM to high-speed modems in the voice frequency band.

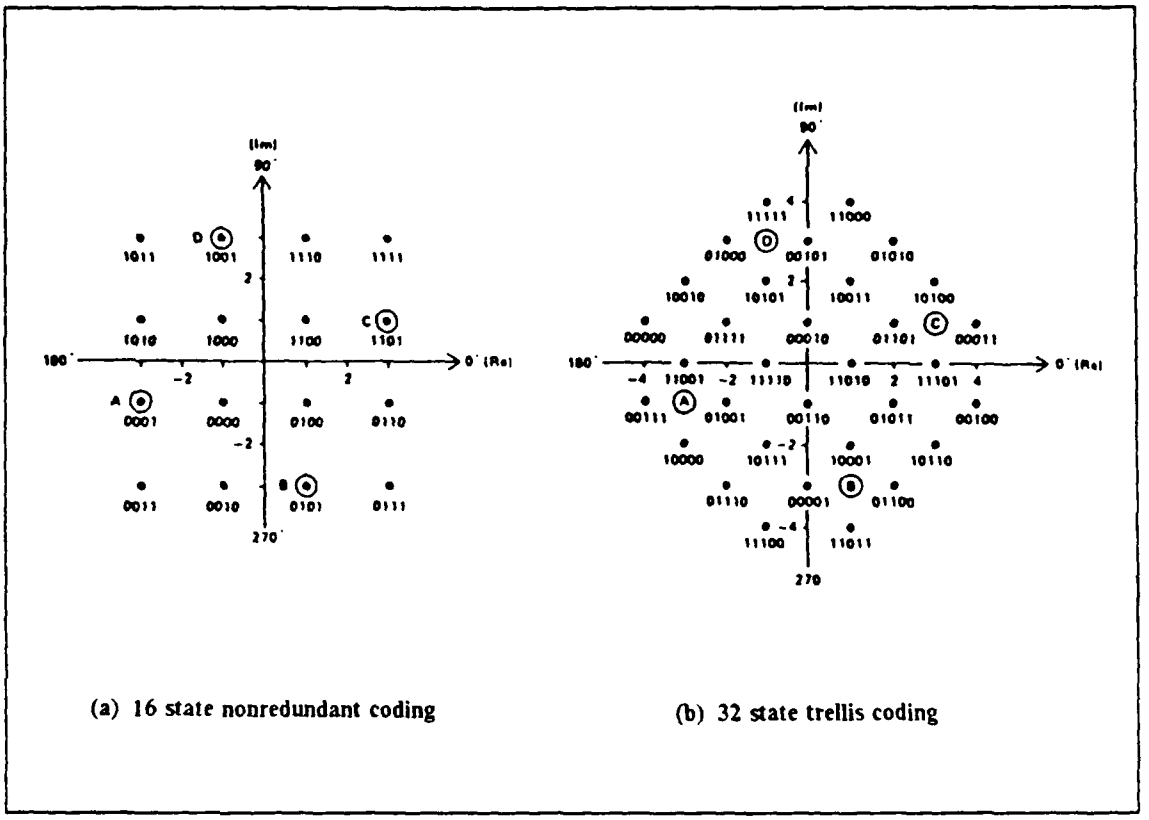
1. V.32 Modem Description

The V.32 modems are intended for use on public switched telephone networks (PSTN) and on point-to-point leased line telephone circuits. These circuits will be discussed in the next section. The following characteristics are common among V.32 modems [Ref. 4 pp. 221-226]:

- Capable of full-duplex operation on the above telephone circuits at 9600 bps.
- Channel separation through the use of echo cancellation techniques.
- Quadrature-amplitude-modulation (QAM) for each channel with synchronous line transmission at 2400 bauds.
- Capable of operating in the following modes:
  9600 bps synchronous,
  4800 bps synchronous,
  2400 bps synchronous.
- At 9600 bps, there are two alternative modulation schemes, one using trellis coding with 32 carrier states and one using nonredundant coding with 16 carrier states.
- The rate sequence is exchanged during start-up to establish the data rate, coding and any other special facilities.

Full-duplex operation means that the modem can transmit and receive data simultaneously on the same frequency on a two-wire or four-wire telephone circuit. This is accomplished through the use of echo cancellation techniques, which will be discussed later in this section. Synchronous transmission means that the transmitted data is always accompanied by a clock signal. The data changes on one edge of the clock and should be sampled by the receiving device on the other edge. Synchronization between the transmitting and receiving modems allows data to be properly timed for receiving and decoding.

The V.32 modems have a carrier frequency of 1800 ±1 Hz, and must be able to operate with received frequency offsets of up to ±7 Hz. The modulation rate of these modems is 2400 bauds ±0.01%. The signal-point constellations used for the two alternative modulation schemes are shown in Figure 2.



(a) 16 state nonredundant coding

(b) 32 state trellis coding

Figure 2. V.32 signal-point constellations: (from Ref. 4: pp. 222,226).

Both of these schemes utilize differential coding. Trellis coding encodes four bits as five, which increases the number of points (states) in the constellation from 16 to 32. Trellis coding, also referred to as forward error control, is actually a way to minimize errors rather than to correct them. This modulation scheme has proven to give a superior signal-to-noise ratio for situations where there is

primarily white noise, rather than burst noise. In situations where there is a lot of burst noise, backward error correcting coding works better. Telephone networks are much more prone to white noise than burst noise, so trellis coding was a good choice for the V.32 modems. The big-trade off for trellis coding is between the amount of data that is saved for decoding and the overall processing time required for operation of the modem. When a receiving V.32 modem loses the signal, it must "retrain" to synchronize with the transmitting modem. Unlike slower modems, where the retrain time can be less than a second, V.32 modems can take up to 10 to 12 seconds to retrain. If the modem has to retrain too often, the throughput can be significantly degraded. For this reason, the timing and trellis algorithms must be carefully thought out in the design and construction of a V.32 modem. [Refs. 4, 5]

As with slower speed modems, V.32 modems contain a self-synchronizing scrambler and descrambler. Unlike slower speed modems which utilize the same generating polynomial for the calling and answering modems, V.32 modems have different generating polynomials for each transmission direction due to the use of the same carrier frequency for transmissions in both directions. The purpose of the scrambler is for randomizing the data sequence, not for encryption or secrecy. Many components in data communication systems work best with random bit sequences, such as the adaptive equalizers and echo cancellers in the V.32 modems. There are sequences of zeros or ones, or periodic sequences that might appear in the data sequence to be transmitted that must be recoded for transmission if the data transmission equipment has trouble in transmitting these sequences. Scramblers recode these undesirable sequences, removing most of the

common repetitions in the input data. The data sequence to be transmitted is formed by the scrambler effectively dividing the input data sequence by the generating polynomial and taking the coefficients of this division in descending order at the output of the scrambler. To recover the message, the received data sequence at the receiving modem is multiplied by the scrambler generating polynomial. The major drawback to the use of scramblers and descramblers is that error performance can be affected. A single error in the transmitted data sequence may cause multiple errors at the output of the descrambler, due to the propagation of the bit error in the shift register of the descrambler. Fortunately, this propagation effect only lasts for a small number of bits. [Refs. 6, 7]

2. Echo Cancellation

Echoes are impairments in the telephone channel which are caused by signal reflections at points in the transmission path where there is a mismatch of circuit impedances. There are several types which will be discussed. Near-end echoes occur at the hybrid of the transmitting modem and at the hybrid of the central office. They are caused by reflections at these points due to mismatches of impedances between the telephone line and the modem's hybrid. This type of echo is predominant and is characterized by a small delay time, typically less than 25 milliseconds (ms). A hybrid is a coupler used to make connections between two-wire and four-wire circuits on the transmission path and in modems to allow the transmitter and receiver to be connected to the telephone line at the same time. Two-wire lines, four-wire lines, and central offices will be discussed in the next section.

Far-end echoes occur at the hybrid of the receiving end central office and the hybrid of the receiving modem. This type of echo is caused by reflections from one of the hybrids at the far end of a four-wire circuit due to impedance mismatches between the four-wire to two-wire connection. The delay time for far-end echoes is higher than for near-end echoes.

Listener echoes occur when a modem's receiver first hears a signal and then hears its echo, and results from the signal making a single reflection. This type of echo is not a problem with modems because the echo is usually much weaker than the original signal. The adaptive equalization of the modem generally removes the listener echo. Talker echo is much more troublesome than listener echo. It is caused by a signal being reflected a second time and results in a modem's transmitted signal being reflected back into its receiver. Talker echo is removed through the use of echo cancellers in the telephone network. [Ref. 5]

The ability of a high-speed modem to perform echo cancellation is a primary factor in its overall performance. There are two general types of echo cancellers. One type is located within the telephone network and the other type is located in echo cancelling modems, such as the V.32. Both types operate similarly in that echoes are cancelled by subtracting an estimated replica of the echo from the signal containing the true echo. When V.32 modems are used over the PSTN, the network cancellers are disabled by the answer tone of the modem, which has periodic phase reversals, unlike the steady tone of most slower speed modems. This phase reversal activates the canceller-disable circuits contained in the network cancellers. The network echo cancellers, also referred to as echo suppressors, detect data transmitted from one end of the connection and

suppresses all signals going the other way. Since many high-speed modems are full-duplex, the network echo cancellers must be disabled so full-duplex communications can take place. The network cancellers are used for modems which are not full-duplex, such as many slower speed modems. The following discussion concerns the type of echo canceller located in the modems.

There are two stages of echo cancelling performed in modems, one cancelling the near-end echoes and one the far-end echoes. Figure 3 is a simplified block diagram of an echo canceller.

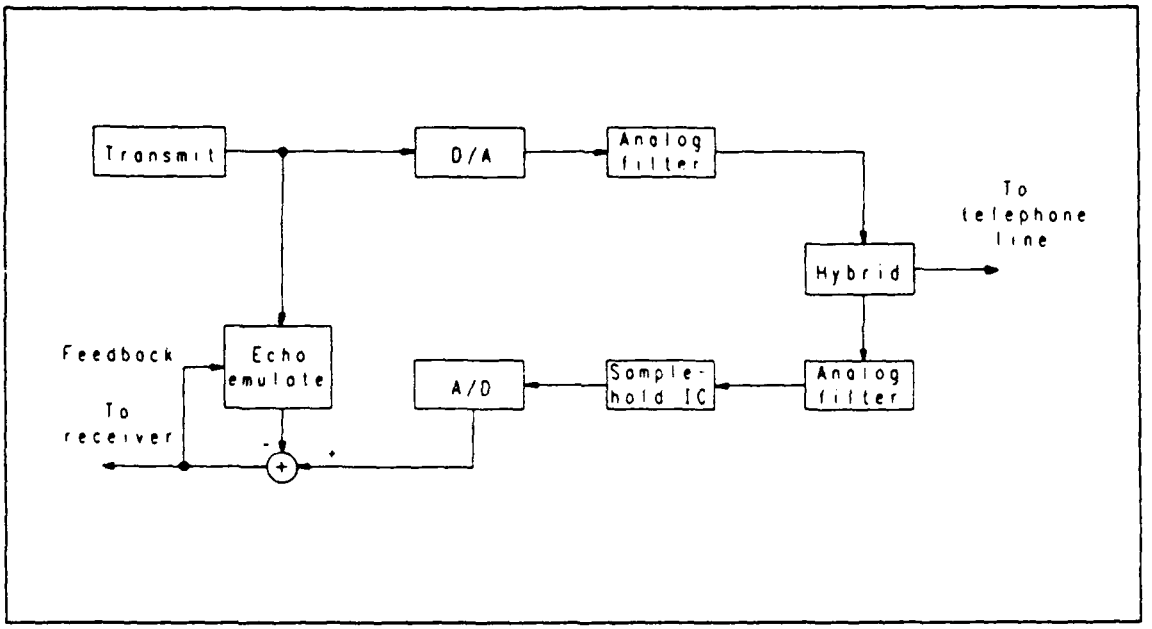


Figure 3. Simplified block diagram of an echo canceller: (after Ref. 5: p. 52.).

The echo emulator block in the diagram contains both the near-end and far-end echo cancellers. These cancellers can be implemented as a passband circuit, where the echo is cancelled before the signal is demodulated, or as a bandpass circuit, where the echo is cancelled after the signal is demodulated. All echo cancellers

contain an adaptive tapped-delay line circuit which dynamically forms an echo replica that is of approximately the same magnitude and phase of the true echo signal. This circuit is the heart of the echo canceller. It allows the replication of an echo signal that is nearly identical to the true echo if it is operating correctly. The transmitted modulated signal (impaired by noise and other interferences from the transmission path) is used as the input to the echo canceller. The echo canceller first estimates the transfer function of the echo and then adaptively updates the information in the canceller to approach the actual response of the path from the original estimate. The initial estimate of the echo transfer function is based on the delays that have been calculated for the system. These delays are calculated during the initial handshaking sequence (defined in V.32 protocol in Ref. 4) that takes place between the two modems during the startup phase. During this sequence, each modem sends a half-duplex signal through the telephone line to find the line's echo characteristics. This information is used to set the taps in the echo canceller's adaptive delay circuit. The output of the adaptive circuit after it has been updated is the replicated echo which is then subtracted from the transmitted, modulated signal (which contains the true echo). The result is checked for correlation with the transmitted signal using a decision-feedback-equalizer type algorithm. If the transmitted and received signals show correlation, the echo emulator of the canceller is modified to cancel the correlation. When the two signals show no correlation, the echo has been maximally removed. [Refs. 8, 9]

Although echo cancellation may sound like a reasonably easy task after the description above, it is very complicated. Accurately detecting the presence

and time delay of echoes is one of the most difficult tasks that is done during echo cancellation. Another factor in effectively cancelling echoes is the ability of the modem to neutralize the effects of any impairments that may affect the signal as it passes through the transmission path such as noise, frequency translation, envelope delay distortion, attenuation distortion, amplitude and phase jitter, quantization effects, and intermodulation distortion. These impairments can distort the echo and cause it to be falsely or inadequately cancelled. The higher the speed of the modem, the more it is affected by these impairments. The modem must be able to perform well in the presence of impairments or it will not perform well on the PSTN. The reader is directed to Refs. 8 and 9 for further discussion concerning neutralization of impairments.

B. DESCRIPTION OF THE VOICE FREQUENCY SWITCHED TELEPHONE LINES

This section provides a description of the PSTN and private telephone network which both use a voice frequency channel in the 0-4000 Hz band for voice and data communication. The actual voice channel used in these systems is 200-3400 Hz for most applications. This 200-3400 Hz voice channel was simulated for this thesis by a bandpass filter which will be discussed in the next chapter. The purpose of this section is to provide the reader with a basic knowledge of the system over which MFM will be applied in the future.

1. Switching Systems

The main question to be answered is, "Why use switching instead of having direct connections between all users?". The cost of such a system and the number of lines and connections would be prohibitive. Switching reduces the

number of lines required dramatically, and multiplexing on the networks reduces the number of lines even more. There are some trade-offs associated with the use of switching. With the addition of switches, the system becomes more complex and with fewer lines the system can become overloaded and blocking may occur if the demand is too high. The network must be expandable to meet the needs of the future, as well as peak traffic periods such as holidays or national emergencies. Large switching centers have the Electronic Switching Systems (ESS), which are capable of terminating hundreds of thousands of lines and of processing hundreds of thousands of calls per hour.

In order to provide a logical and efficient means to switch, a hierarchy was established. There are two basic types of switches in the PSTN, local offices and toll offices, of which there are four levels. The local office or central office (CO) is the lowest switch in the network. There are over 20,000 COs in the network. The user is directly connected to the CO through a transmission link. Many COs are connected through transmission links to a single switching office, called a toll center, which is located on the lowest level of the toll network. The three higher levels of switching in the toll network are called primary centers, sectional centers, and regional centers (the highest level of the PSTN). Ten regional centers are located in the United States and two in Canada. The lowest available level of the PSTN is always used for routing traffic. This is done because fewer network facilities are used resulting in shorter transmission paths, ultimately resulting in better circuit quality.

In addition to the five levels of switching in the PSTN, there is one more type of switching system in use, called a private branch exchange (PBX). The

PBXs are not part of the PSTN, but do provide the users access to the PSTN through a transmission link to the COs, as well as performing internal switching functions for the users of the private telephone network. The private telephone network consists of the PBXs and the transmission links connecting the users to the PBXs. This system is available to users through leasing of the lines.

There are two types of user-to-user connections that may be established on the telephone networks. The first, the dialed circuit, or dial-up line, is a switched circuit telephone line connection established on the PSTN. This is the type of connection most users establish on a day-to-day basis. The quality of a dialed circuit can vary widely and is difficult to predict. One connection between two points may have an excellent quality, while another connection between the same two points may be terrible. This difference in quality of the connection is because the transmission path will most likely be different from one call to the next. The second type of connection, the private leased line, is a line that connects two or more communication points on a dedicated 24 hr/day basis. The characteristics of this type of connection are guaranteed to meet certain criteria. There are two different sets of criteria specified for leased lines:

- *C-conditioning*. Specification of the frequency response and envelope delay (linear distortion) characteristics of the line.
- *D-conditioning*. Specification of the minimum signal-to-noise ratio (SNR) and the second- and third-harmonic (nonlinear) distortion minimum signal-to-distortion ratios. [Ref. 10: pp. 691-692]

There are three types of switching presently in use in the telephone system: circuit switching, message switching, and packet switching. Circuit switching is used for voice and data communication and is the predominant type. It is

accomplished by establishing a dedicated path for the duration of the call. Not to be confused with a leased line, this dedicated path is only for the duration of the one call; the next call between the same two points will be on a dedicated path, but most likely a different path. Circuit switching is most efficient for calls of long duration and can use three types of multiplexing: space division multiplexing (SDM), frequency division multiplexing (FDM), and time division multiplexing (TDM). Message switching is used for transmitting data only, and uses only SDM. With this type of switching, the entire message is stored into memory at each switching station and then forwarded to the next station as the message traffic load permits. It is routed to the destination listed in the header information of the message. Message and packet switching are both known as store-and-forward switching, because the messages and packets are stored into memory at each switching station and then forwarded as the load permits.

Packet switching is the latest technology; it is readily adaptable to digital processing and uses only TDM. This type of switching is uniquely different from circuit and message switching in that the data is broken down into segments called packets and then sent via the first open line to the destination. As in message switching, the packets are stored into memory at each switching station. With packet switching the transmission channel is only occupied during the transmission of each packet. The packets may or may not follow the same path and consequently may arrive out of order. When the packets arrive at the destination they are sequenced and processed in order. Packet switching is the most efficient type of switching when the duration of the call is relatively short compared to circuit or message switching, because of the extra overhead required

for routing, packet construction, and sequencing. MFM is well-suited for packet switching.

2. Transmission Lines

The transmission links between the users and the switching centers are referred to as trunks. These trunks can be implemented with a variety of mediums including twisted pair (pairs of wire), coaxial cable, point-to-point microwave links, and optical cable. Most of the telephone network still uses twisted pair, so this is the medium that is discussed here. Transmission through a single wire (with a ground return) is possible and has been used in the past, but the noise level of the circuit is unacceptable for customer use. The twisted pair (known as two-wire) presently used is a balanced pair of wires through which signals propagate as a voltage difference between the two wires. Interference or induced noise is coupled equally into both wires of a twisted pair and propagates along the pair in one direction. Almost all user-to-CO trunks in the PSTN are on two-wire links. The two-wire link allows for two-way communication. The trunks used for transmission between switching centers and over longer distances usually involves a pair of two-wire lines, one for transmitting and one for receiving, in which the two connections are to be kept separate. This configuration is referred to as a four-wire system. These systems are often used with some form of multiplexing to provide multiple channels in one direction on one pair of wires.

Since most toll network circuits are four-wire, the switches for these systems are designed to connect both directions of transmission separately. Two paths are needed for each connection for these switches. The switches for

two-wire systems, as used in local switching in the COs, require only one path through the switch for each direction.

C. VOICE FREQUENCY MFM ENCODING SCHEMES

This section provides a brief description of the two encoding formats utilized for this thesis. Bit error rate (BER) and SNR data generated for both encoding formats is discussed in Chapter IV.

1. DQPSK

DQPSK encoding is similar to QPSK encoding in that they both use the same four Gray-encoded two-bit symbols in the signal constellation. Phase ambiguity is eliminated in QPSK through the use of strict phase coherent regeneration of the sampling frequency, but this results in the requirement of complex synchronization techniques. DQPSK resolves the phase ambiguity problem by transforming the original two-bit symbol into a new differential two-bit symbol, which is then encoded as QPSK. This transformation is shown in Figure 4. As can be seen in the figure, the inputs generate new symbols as shown below.

- An input of '00' produces a new symbol in the same quadrant as the previous symbol.
- An input of '01' rotates the new symbol $+\pi/2$ radians from the previous symbol.
- An input of '10' rotates the new symbol $-\pi/2$ radians from the previous symbol.
- An input of '11' rotates the new symbol π radians from the previous symbol.

Decoding of the MFM signal in the receiver is performed by determining the phase difference between successive symbols. [Ref. 2: pp. 9-11]

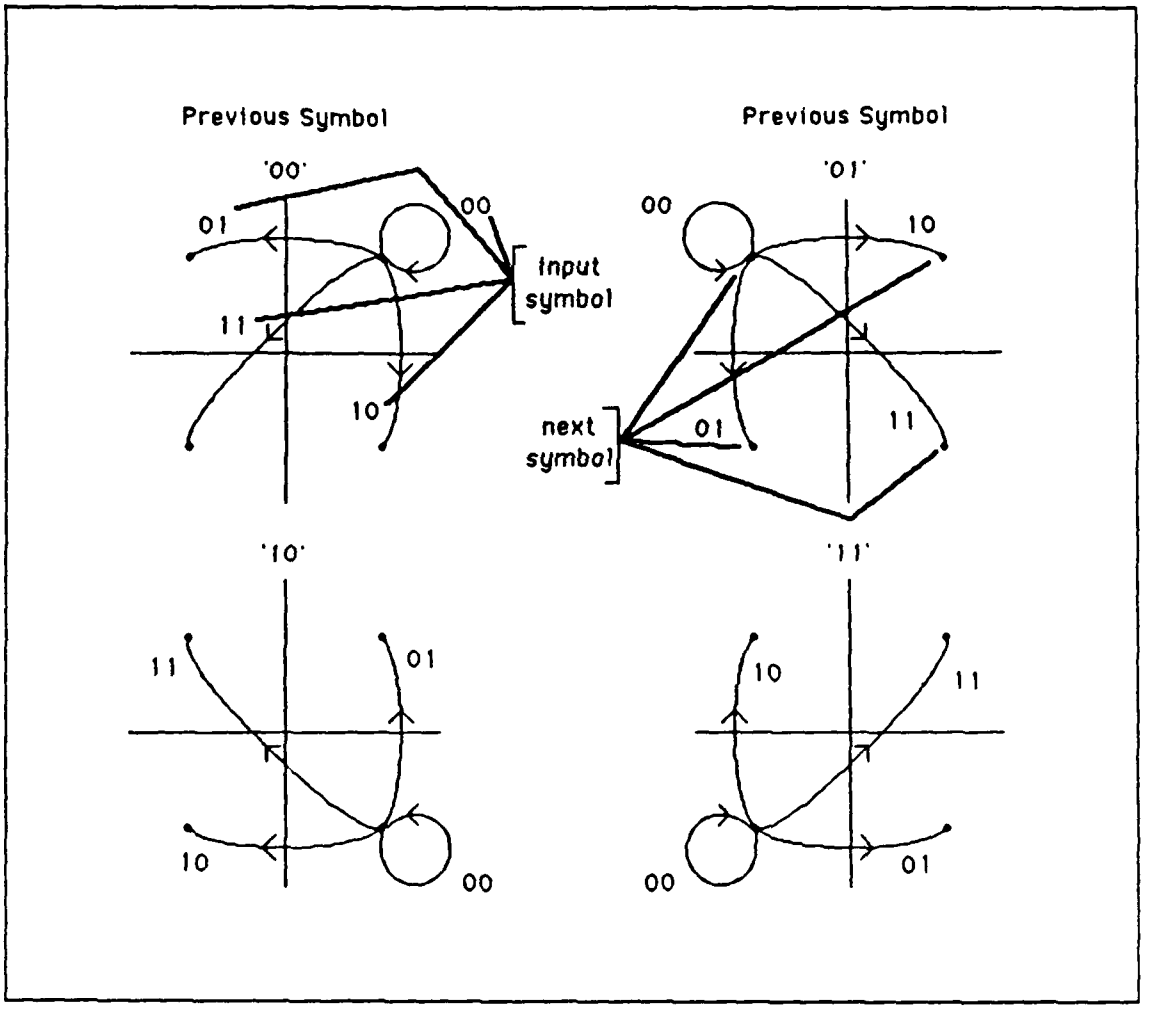


Figure 4. DQPSK encoding scheme: (from Ref. 2: p. 10.).

2. D16-QAM

The data rate for D16-QAM is double that of DQPSK; it uses four-bit symbols vice the two-bit symbols of DQPSK. The Gray-encoded D16-QAM signal constellation studied in this thesis is shown in Figure 5. As can be seen in the figure, this signal constellation has eight 45° sectors and two magnitude levels. Although the data rate for D16-QAM is doubled, it is at the cost of an increased

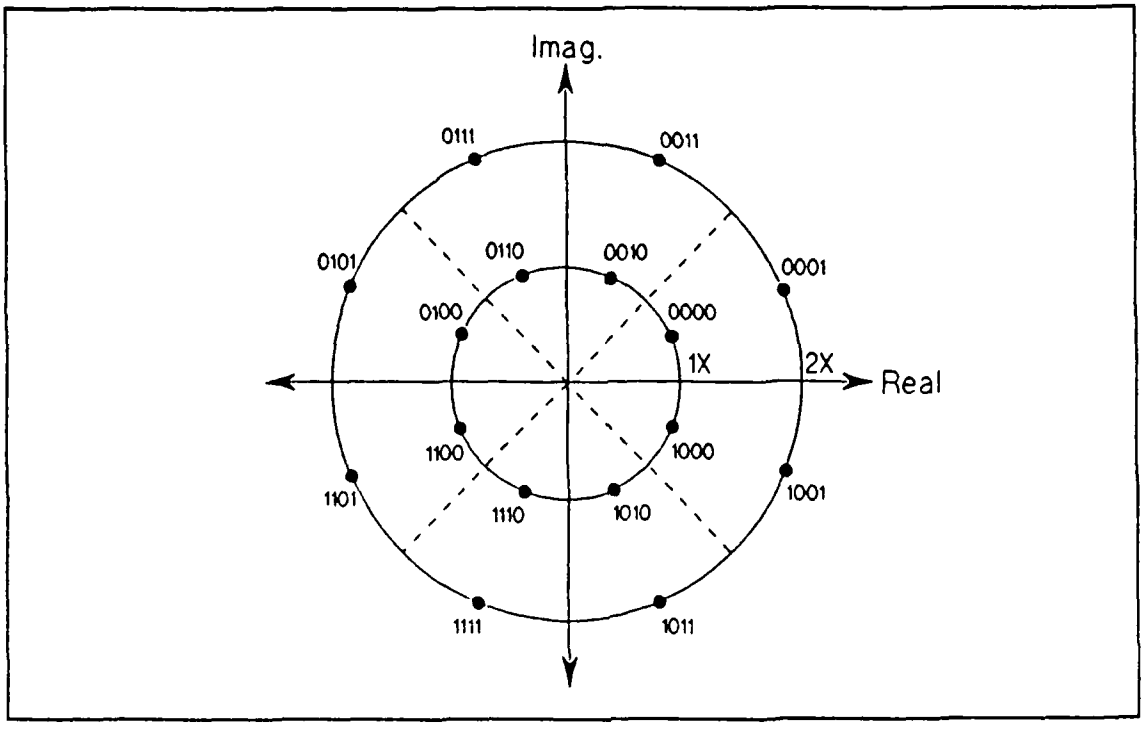


Figure 5. D16-QAM signal constellation.

bit error rate. For the same baud size and noise level, the D16-QAM encoding format cannot accept as much system differential phase error as the DQPSK scheme, due to the smaller sector size. In addition, noise and other system impairments can cause a magnitude error when using the D16-QAM scheme. A detailed discussion of D16-QAM encoding/decoding is contained in [Ref. 3: pp. 10-13].

III. SYSTEM IMPLEMENTATION

A. HARDWARE

The MFM system developed by LT Terry K. Gantenbein, USN, in Ref. 2 and modified by LT Peter G. Basil, USCG, in Ref. 3 is the basis for the experimental part of this thesis. The hardware utilized in this research is the same with the exception of the channel filter. The 16-20 kHz bandpass filter utilized by Gantenbein and Basil was replaced with a 200-3400 Hz filter for the testing described in Chapter IV.

1. MFM Transmitter

The MFM transmitter contains a D/A converter circuit that is built on an IBM PC/XT interface breadboard which is inserted into an expansion slot in the transmit PC. The original transmitter, designed and built by LT Robert D. Childs, USN [Ref. 11], was modified by Gantenbein [Ref. 2]. The current configuration of the transmitter is documented in the schematic in Appendix A. A second MFM transmitter board was built during this thesis to allow flexibility by providing a backup transmitter for testing alternate filtering schemes. The transmitter board used in this thesis for all data transmission was this backup transmitter.

A functional block diagram of the transmitter is given in Figure 6. The input to the encoder is a serial binary signal which is processed through the transmitter as discussed in Chapter I. The encoding of the input data is

accomplished in software and is discussed in the next chapter. The output of the D/A converter contains MFM tone frequencies in the 200-3400 Hz band.

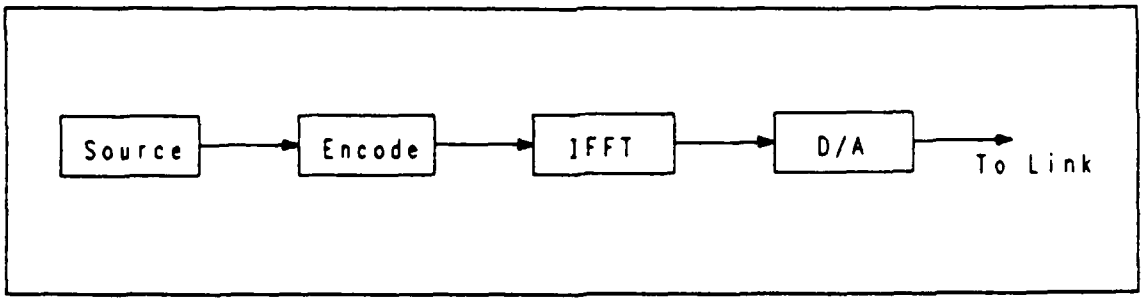


Figure 6. Block diagram of the MFM transmitter.

2. MFM Receiver

The MFM receiver utilizes a Metrabyte, Inc., DASH16F data acquisition board with software routines to accomplish the A/D conversion. The reception/demodulation of the MFM signal is the reverse of the transmission/modulation process as discussed in Chapter I. The receive PC utilizes a PL1250 floating point processor (FPP) board from Eighteen-Eight Laboratories, Inc., to perform the FFT on the output of the A/D converter for decoding. This allows real-time demodulation of the signal as described by Basil [Ref. 3]. A functional block diagram of the receiver is given in Figure 7.

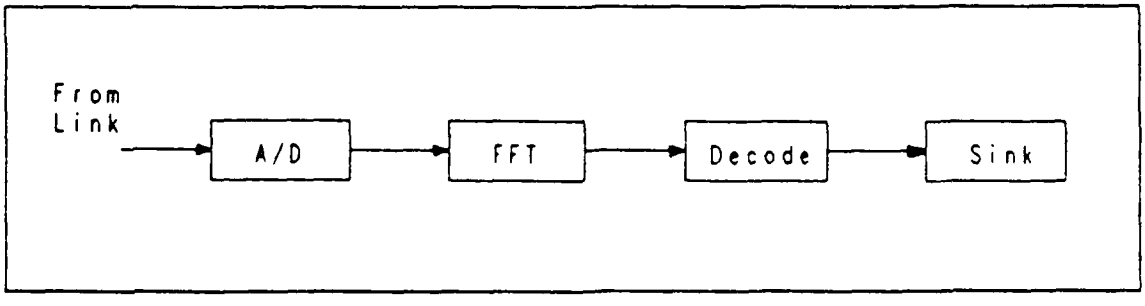


Figure 7. Block diagram of the MFM receiver.

3. MFM Synchronizer

In MFM communications, each transmitted packet must be synchronized at the receiver before the signal can be acquired and demodulated. Each transmitted packet has a synchronization baud of length 256 appended to the beginning of packet. The polarities of the synchronization baud are known. The receiving PC contains a synchronization board which is pre-loaded with the known polarities of the last 128 samples of the synchronization baud for the purpose of synchronization. The synchronization process and the board designed and built to accomplish synchronization are discussed in detail by Basil [Ref. 3].

4. Voice Channel Filter

The 200-3400 Hz channel characteristic of the public switched telephone network was simulated by a National Semiconductor TP3040J PCM monolithic filter chip. The 16-pin TP3040J contains both a transmit filter and a receive filter designed to the specifications used in the telephone network. The transmit filter is a fourth-order Chebyshev highpass filter in series with a fifth-order elliptic lowpass filter. The receive filter is a fifth-order lowpass filter designed to reconstruct the voice or data signal from a decoded signal. Both filters are constructed with switched capacitor integrators. [Ref. 12]

The master input clock frequency for the TP3040J can be selected as 2.048 MHz, 1.544 MHz, or 1.536 Mhz. A clock frequency of 2.048 MHz was used for this thesis. The gain of the filter is also selectable and can be chosen by properly selecting the resistor values. The gain was chosen to maximize the

amplitude of the signal, without exceeding the range of the A/D converter, which was set at ±2.5 volts. The TP3040J was connected as shown in Figure 8.

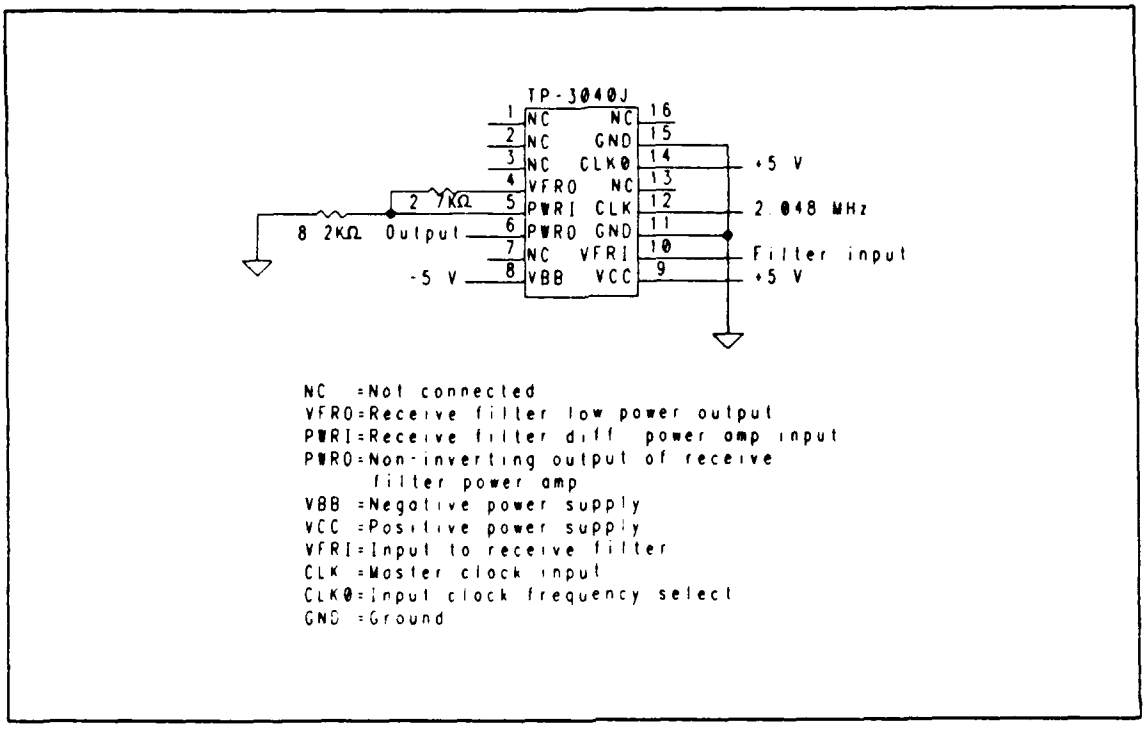


Figure 8. Voice channel filter wiring schematic.

B. SOFTWARE

The programs used in this thesis research are written and compiled in Turbo Pascal, version 5.5, from Borland International. The programs are similar to those written by Gantenbein for DQPSK, but have been modified for application to the voice frequency channel and for D16-QAM encoding/decoding. The modified DQPSK and D16-QAM programs used for this research are included as Appendices B, C, D, and E.

1. Transmitter

The DQPSK and D16-QAM transmit programs are called DQPSKXMIT and DQAMXMIT respectively. The programs are divided into subroutines, known as procedures in Pascal. The procedures are discussed in the order in which they are called in the programs. The primary difference between the two programs is in the encoding section of the programs, due to the different modulation constellations used. Unless otherwise noted, the following discussion applies to both programs.

The PL1250 FPP is initialized at the start of the main body of the transmit programs. Next, the procedure SYNCBAUD generates the synchronization baud (syncbaud) of length 256 of which the last 128 samples are used for synchronization with the receiver. The polarities of the syncbaud samples are identical each time SYNCBAUD is executed, so the receiver can acquire the transmitted data as long as the synchronizer board has been properly loaded with the syncbaud. SYNCBAUD calls the procedure CNVTTOTIME which, in turn, calls the PL1250 FPP to perform the IFFT on the syncbaud tone values. Next, SELECTBAUD chooses the values of k_1 and k_2 based on the baud length selected by the user. The procedure TAILORPACKET then determines the maximum number of bauds that can be transmitted for each packet. Next, DIFFENCODE performs the encoding of the data as described in Chapter I, using the different encoding formats for DQPSK and D16-QAM. DIFFENCODE also calls CNVTTOTIME to perform the IFFT on the data. The next procedure called is SCALEDATA, which takes the time domain samples and stores them in an array of 61440 bytes. If the message is longer than 61440 bytes, then the entire message

cannot be sent at once. DMAINIT takes the values stored in the array and clocks them through the D/A converter. Finally, DMASTOP ends the data transfer when all values in the array have been clocked through the converter. The DQPSK and D16-QAM transmit programs are included as Appendix B and Appendix C, respectively.

2. Receiver

As with the transmit programs, the DQPSK and D16-QAM receive programs are very similar to each other; both are discussed in this section. The two receive programs are called DQPSKREC and DQAMREC. The procedures in the receive programs are discussed in the order in which they are called, and unless otherwise noted, the discussion applies to both receive programs.

The PL1250 FPP is initialized at the start of the main body of the receive program and a pointer is set up by GETDMABUFFER to point to the array in which the received values are stored. PACKETSETUP, the first procedure called, determines the maximum number of bauds that can be received and assigns the values of k_1 and k_2 based on the baud length input to the receive PC by the user. The receive baud length must be the same as the baud length of the transmitted data. The next procedure, ACQUIREDATA initializes the DASH16F board for the 12- bit A/D conversion process. The received data is stored in the DASH16F board, in 16-bit words. The four most significant bits of each word contain channel identification data, so the procedure CONVERTDATA removes these four bits. The remaining 12-bit words are sent to the main body of the program. The PL1250 FPP performs the FFT of the data and places the resulting values in the first half of the complex array as discussed in Chapter I. The next

procedure, DIFFDECODE, performs the decoding of the data. In DQPSKREC, DIFFDECODE decodes the phase difference between adjacent tones for all transmitted tones. In D16-QAMREC, DIFFDECODE decodes the phase and magnitude differences between the adjacent symbols for all transmitted tones. The decoded symbols are displayed to the receive PC monitor as ASCII characters, with two four-bit symbols representing one ASCII character. Each baud of the received message is processed by the receive program until the whole received message has been displayed. The receive programs are included as Appendix D (DQPSKREC) and Appendix E (DQAMREC).

3. Synchronizer

The synchronizer board is initialized with the program SYNCLOAD which calls an assembly language routine named SYNCINIT. SYNCLOAD causes the synchronizer board to be initialized with the last 128 samples of the syncbaud. The synchronizer board needs only to be initialized once each time the MFM system is powered up, prior to transmitting any data. The syncbaud values are contained in the data file VALS.DAT and are loaded into the synchronizer board by typing "SYNCLOAD" and then pressing the enter key on the receive PC keyboard. The program SYNCLOAD is included as Appendix F.

IV. SYSTEM TESTING AND RESULTS

This chapter discusses the performance evaluation of the MFM system over a voice frequency channel. The first section deals with the system phase response and the selection of the syncbaud delay to optimize the system performance. The second section discusses the SNR and the bit error rate estimates for various input SNRs.

A. SYSTEM PHASE RESPONSE

The relevant system phase response is the difference between the received phase and the transmitted phase for each tone. The values used for the calculation of phase response are the received phase before decoding and the transmitted phase after encoding. Ideally, the plot of phase response would be flat. The DQPSK phase response plot of the system is shown in Figure 9 for three different delays of the syncbaud. The curve for a delay of one is the most linear and flattest. Figure 10 is a plot of the difference between adjacent received/transmitted tones for DQPSK. Since the sectors for DQPSK are 90° in width, all points within ±45° represent symbols which are correctly decoded. Any points outside ±45° represent symbols which are incorrectly decoded. Figure 11 is the D16-QAM phase response for three different delays of the synchr ization baud. A delay of one produced the most linear, flattest response. The difference between adjacent received/transmitted tones for D16-QAM is shown in Figure 12. The sectors for this encoding format are 45° in width, so all points within ±22.5° represent data which is correctly decoded.

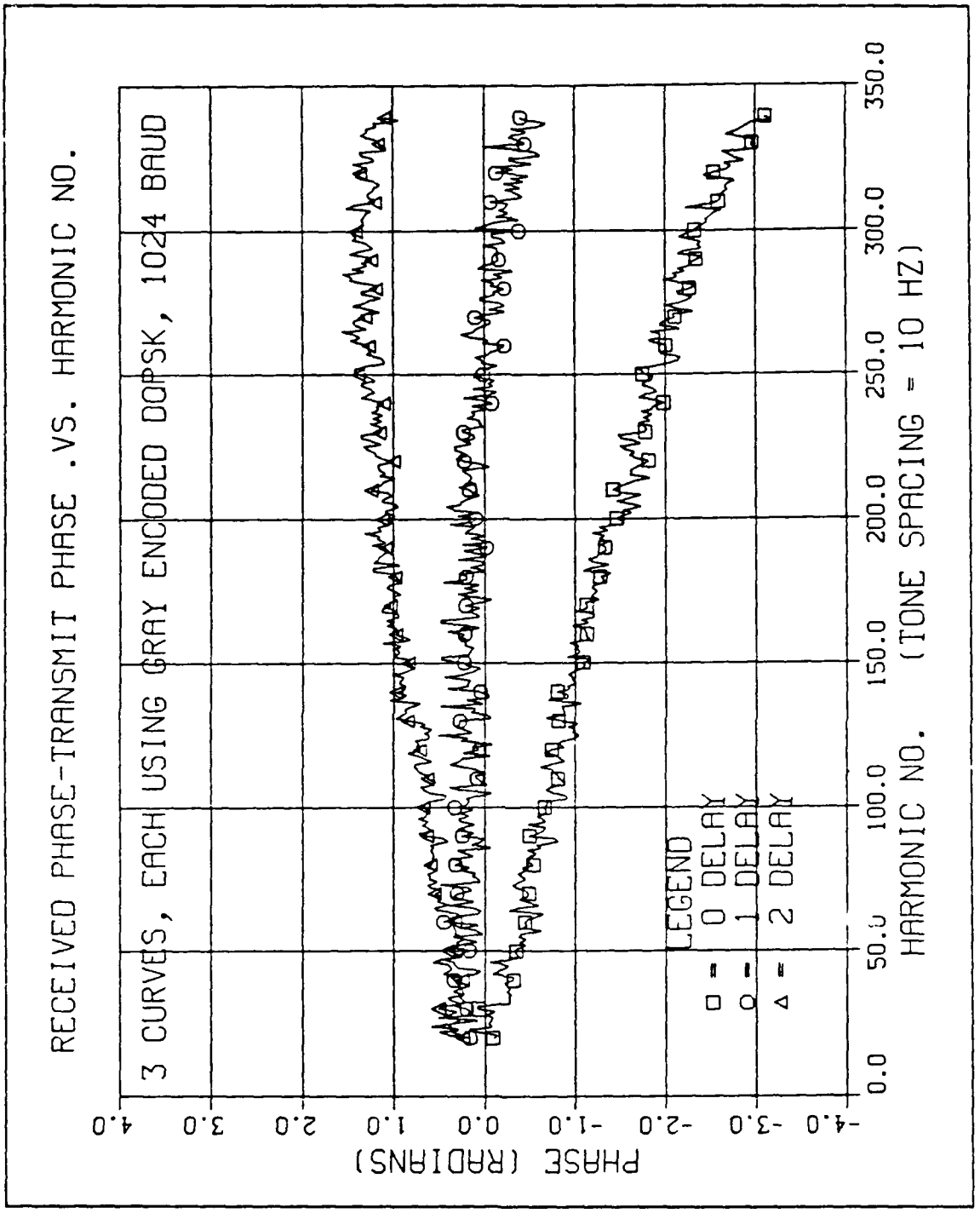


Figure 9. DQPSK phase response for different delays.

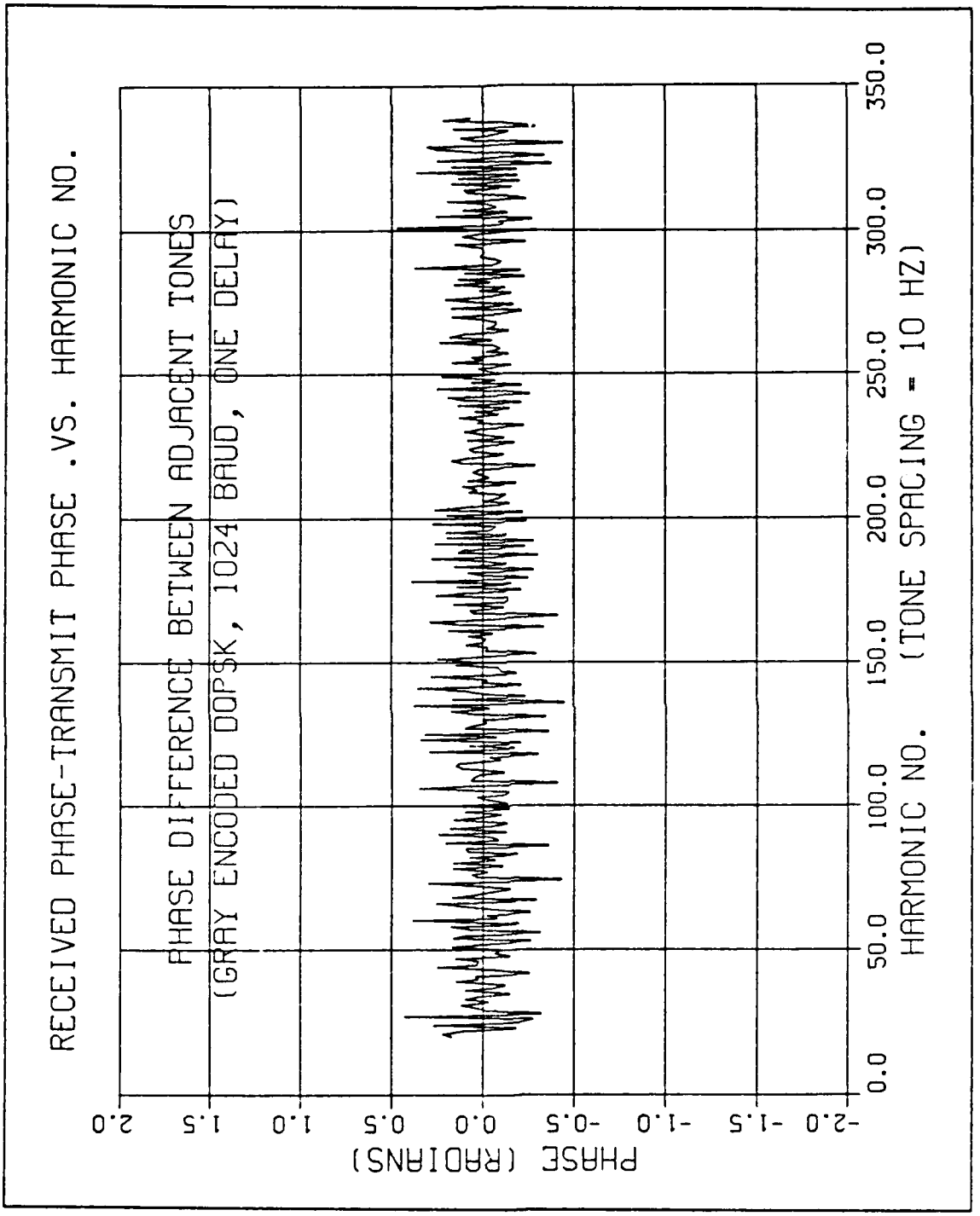


Figure 10. DQPSK phase difference between adjacent tones.

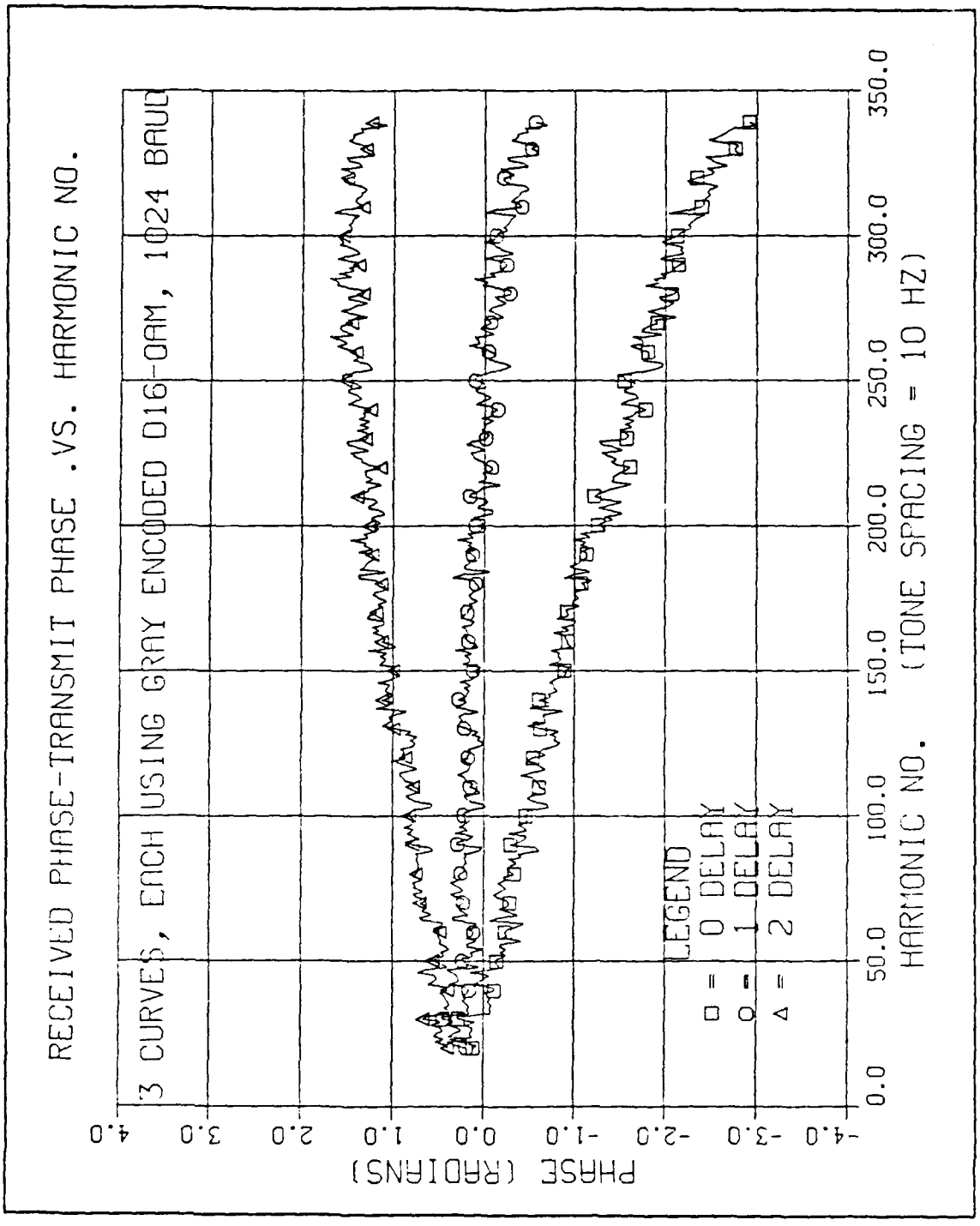


Figure 11. D16-QAM phase response for different delays.

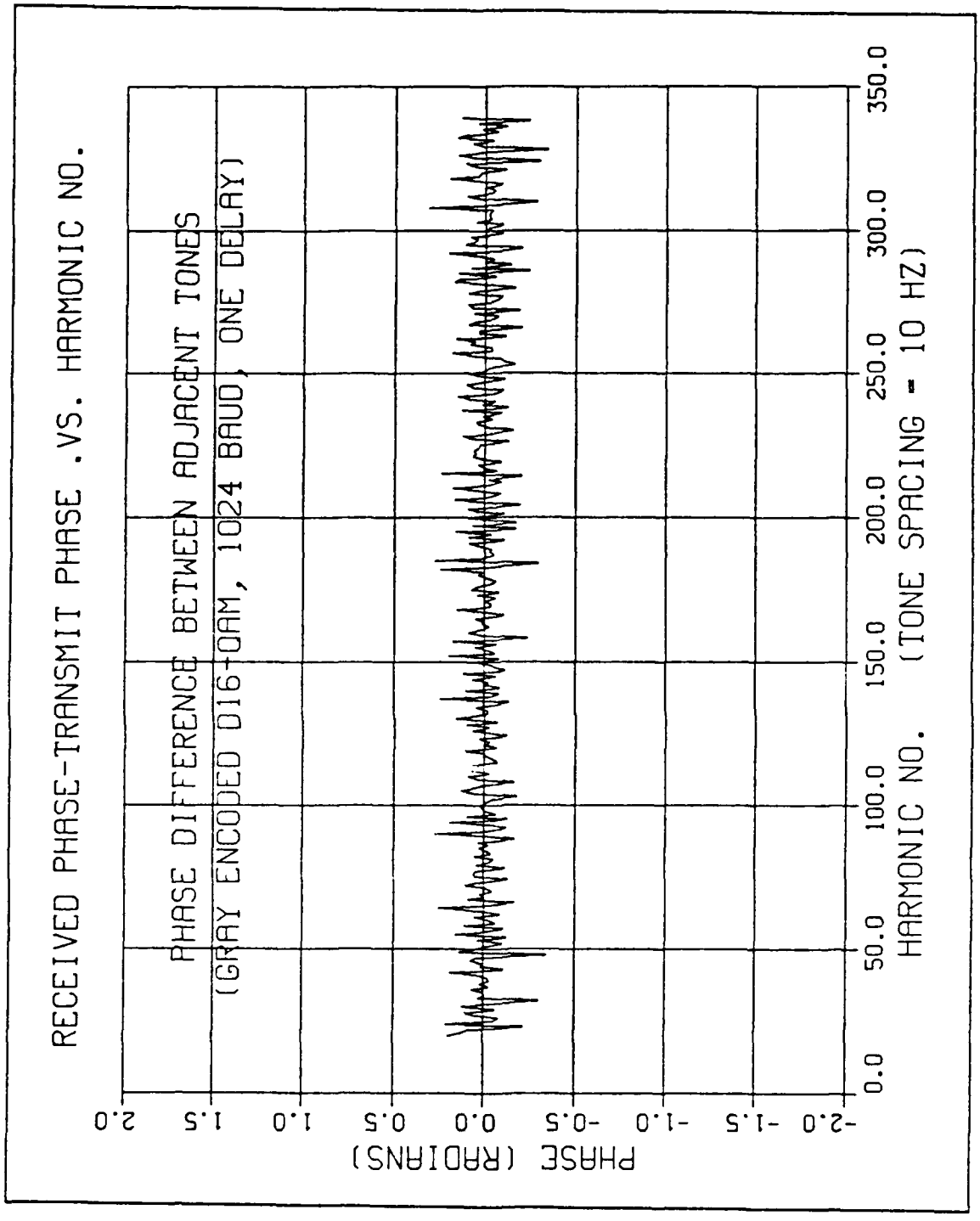


Figure 12. D16-QAM phase difference between adjacent tones.

The phase difference between adjacent tones is introduced by the system and varies from tone to tone. The phase differences introduced by the system increase the possibility that the phase difference between adjacent tones will be incorrectly decoded. Additionally, when noise is added to the system, the phase differences increase, possibly resulting in decoding phase errors. To minimize the possibility of the errors in decoding of phase, sync delays of one for DQPSK and one for D16-QAM were chosen for all subsequent testing which was conducted for this research.

B. BIT ERRORS AND SNR

Bit errors and SNR are both discussed in the same section as they are directly related. The number of bit errors that can be expected in a transmission is directly related to the SNR as will be shown graphically later in this section. The results for DQPSK and D16-QAM encoding formats are discussed separately and will be compared and commented on in the next chapter.

System performance testing was conducted on a channel with additive white Gaussian noise (AWGN), i.e., each transmitted sample is affected independently by the noise. The output SNR was estimated for different input noise levels for baud lengths of 256, 512, 1024, and 2048. The output SNR is defined as the ratio of the square of the mean of each of the complex multiplied adjacent tones to their variances.

1. DQPSK Performance

Data was generated and analyzed for DQPSK encoding of approximately 10,000 bits for each baud length and each input SNR. The program QPSKSNR, included as Appendix G, counts the number of bit errors and estimates the

output SNR for the transmitted data. The system SNR, which is the output SNR of the system with no input additive noise, is shown versus the frequency spacing Δf in Figure 13. As shown in the figure, the performance of DQPSK improves with decreasing Δf . This is as expected, because the phase difference between adjacent tones gets smaller as Δf gets smaller.

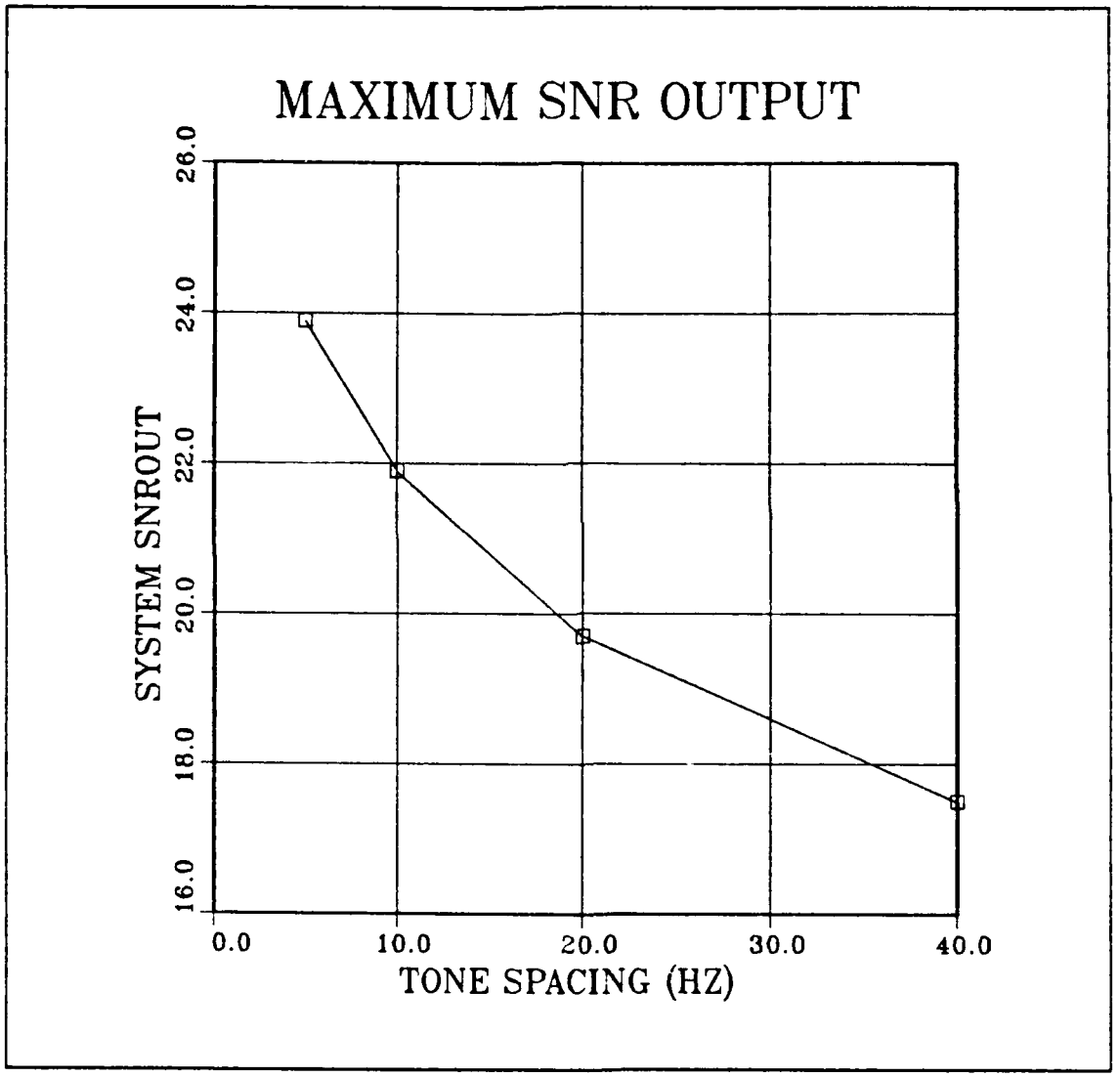


Figure 13. DQPSK system SNR output.

In Figure 14, the output SNR is shown versus various input SNRs for the different baud lengths. The theoretical curve for output SNR versus input SNR is also shown for comparison. According to theory, the output SNR is expected to be equal to the narrowband input SNR [Ref. 1: pp. 24-25].

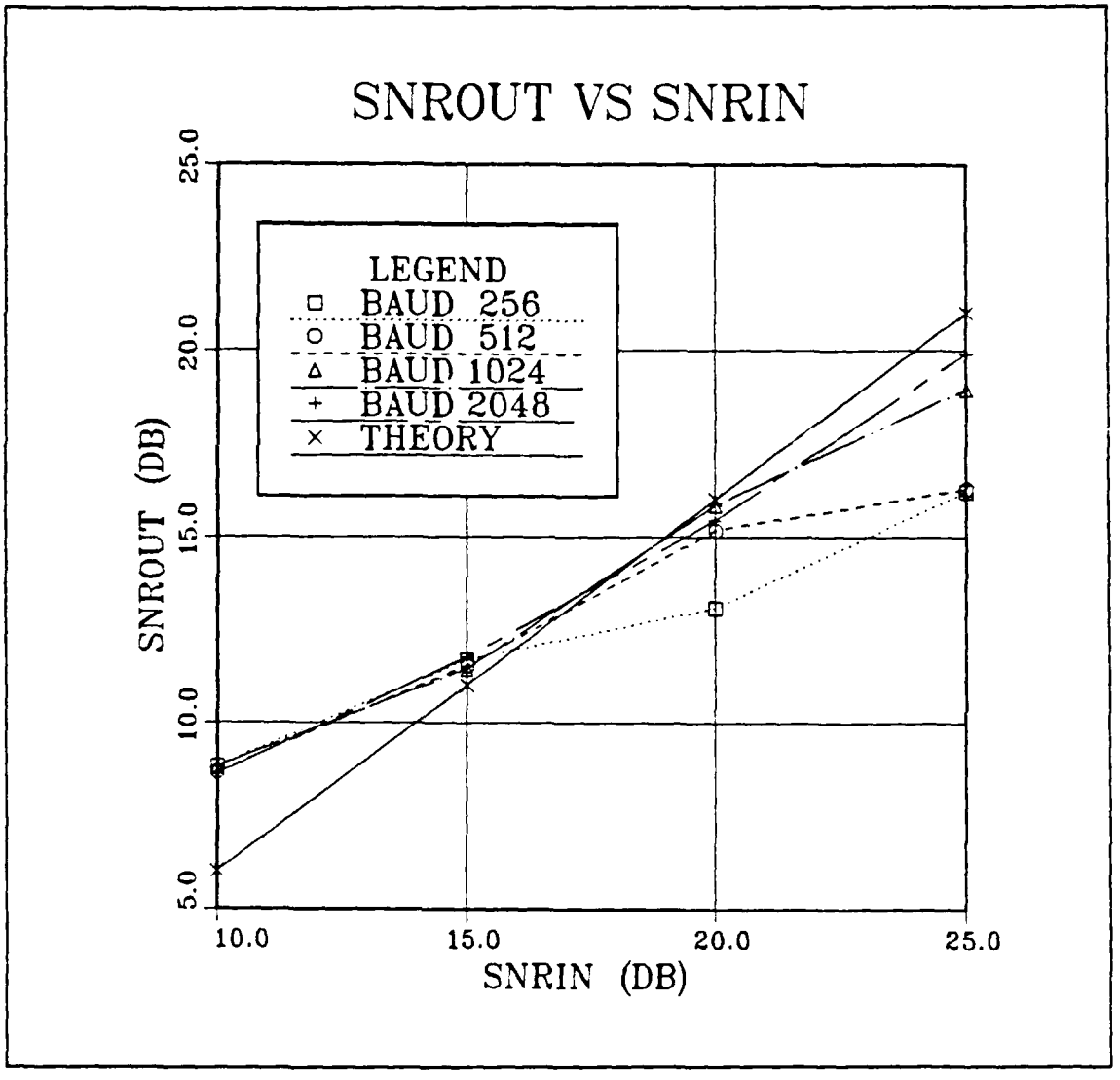


Figure 14. DQPSK output SNR versus input SNR.

Table 2 lists the bit errors for various input SNR levels for the different baud lengths. In general, at higher input SNR levels, the larger baud sizes perform better because the tones are spaced closer together.

Table 2. BIT ERRORS IN 10,000 BITS TRANSMITTED VS BAUD TYPE AND SNR

Δf	k_x	SNRIN (dB)				
		5	**10**	**15**	**20**	**25**
40	256	507	105	43	8	0
20	512	487	57	7	0	0
10	1024	402	42	0	0	0
5	2048	419	27	0	0	0

2. D16-QAM Performance

The data generated and analyzed for D16-QAM encoding consisted of approximately 20,000 bits for each baud length and each input SNR. The number of bit errors and calculation of the output SNR were generated by the program QAMSNR, which is included as Appendix H.

The decoding procedure for D16-QAM, like DQPSK, requires the complex multiplication of adjacent tones to obtain the phase differential. Unlike DQPSK, which has one magnitude level as a result of the complex multiplication, D16-QAM has three magnitude levels as a result of the decoding procedure. In D16-QAM, the encoded transmitted data has two possible magnitudes, dependent upon whether the symbols are from the inner (small) or the outer (large) ring of the constellation. When decoding the received data, the three possible magnitude levels are created as follows:

- Small times small ⇒ Lowest magnitude
- Small times large ⇒ Middle magnitude
- Large times large ⇒ Highest magnitude

These three magnitude levels are affected differently by AWGN. The lowest magnitude level is degraded most by AWGN and the highest magnitude level is degraded least, since the AWGN will affect each magnitude level equally. Figure 15 is a plot of output SNR versus input SNR for different baud lengths. The output SNRs contained in this figure are only for the lowest magnitude level, since it is the one most affected by AWGN. The performance for the other two magnitude levels is better. As with DQPSK, the larger baud sizes in D16-QAM perform better due to the smaller frequency spacing between adjacent tones. The theoretical curve for output SNR versus input SNR for the lowest magnitudes is also shown for comparison and is calculated by the equation [Ref. 13]

$$SNR_{out} = SNR_{in}\left(\frac{2}{c+1}\right), \tag{4}$$

where c is the ratio of the highest magnitude to the lowest magnitude.

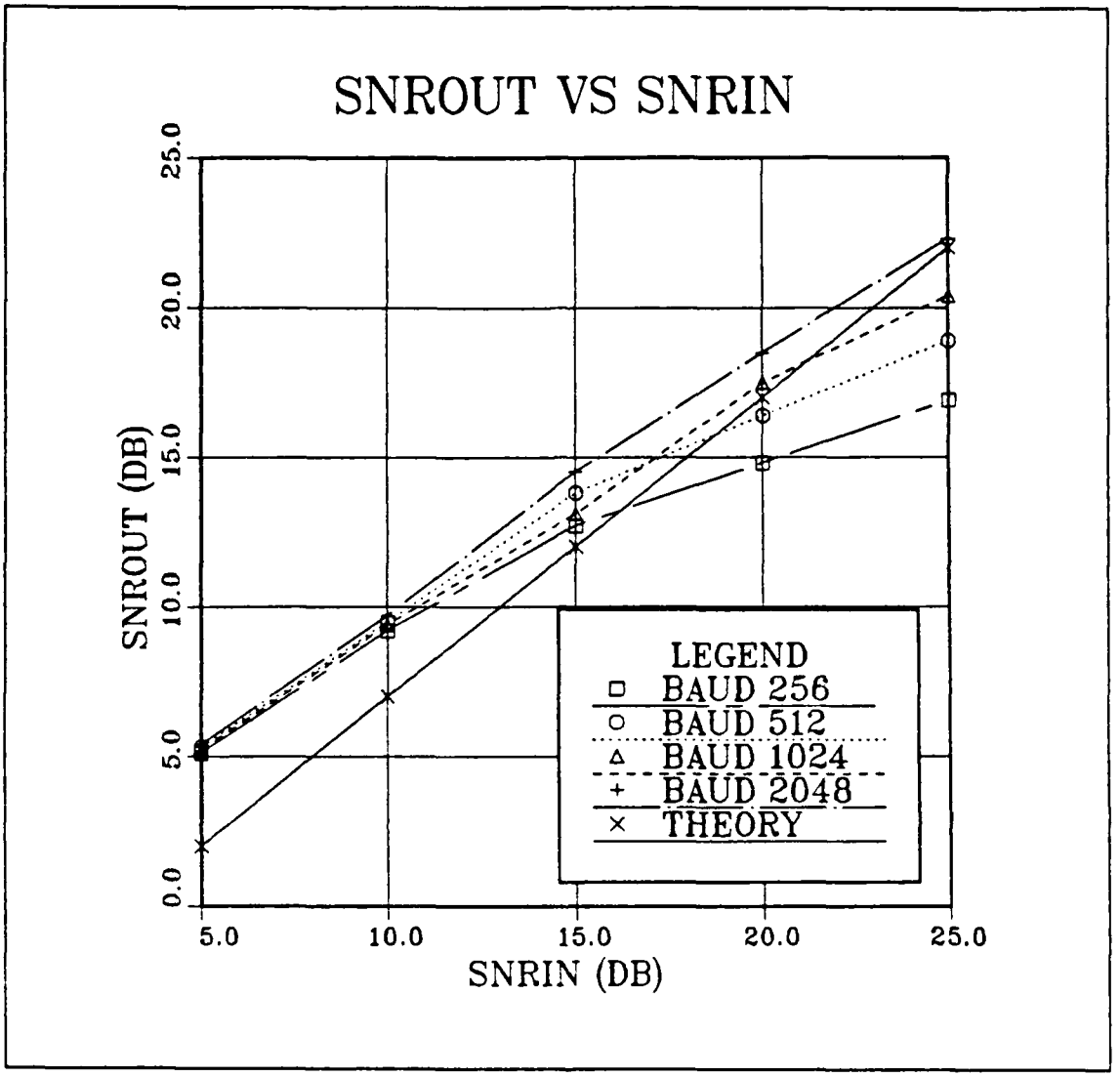


Figure 15. D16-QAM output SNR versus input SNR.

Table 3 shows the number of magnitude and phase bit errors for 20,000 transmitted bits for each of the baud lengths at various input SNRs. As expected, the system performance is better for higher input SNRs and larger baud sizes. In general, the number of magnitude decoding errors and phase decoding errors is approximately equal for almost all cases. Also, most of the phase errors are of

the single bit type, indicating that the phase decoding errors occurred due to received phases being decoded only one sector away from the transmitted sector (since the data is Gray-encoded).

Table 3. MAGNITUDE AND PHASE BIT ERRORS FOR D16-QAM FOR 20,000 BITS TRANSMITTED

Baud Size	256	512	1024	2048
SNRIN (db)	**25**			
Mag bit errors	72	19	5	1
1 bit phase errors	86	11	5	1
2 bit phase errors	0	0	0	0
3 bit phase errors	0	0	0	0
SNRIN (db)	**20**			
Mag bit errors	253	63	42	36
1 bit phase errors	311	45	35	36
2 bit phase errors	7	0	0	0
3 bit phase errors	3	0	0	0
SNRIN (db)	**15**			
Mag bit errors	495	352	379	353
1 bit phase errors	537	393	421	376
2 bit phase errors	8	6	0	0
3 bit phase errors	0	0	0	0
SNRIN (db)	**10**			
Mag bit errors	1114	1035	949	1029
1 bit phase errors	1424	1214	1173	1171
2 bit phase errors	55	13	0	0
3 bit phase errors	5	3	0	1

V. CONCLUSIONS AND RECOMMENDATIONS

With proper choice of synchronization delay, both differential MFM encoding formats performed well in the test environment. The number of bit errors for baud sizes of 512 and larger were acceptable for SNRs above 15 dB, which is 10 dB below the SNR specifications for the PSTN and the private telephone networks.

The data throughput that can be achieved by MFM signals over a voice frequency channel compares favorably with that of high-speed modems. The DQPSK encoding format carries two bits per tone per baud, resulting in a bandwidth efficiency of two bits/s/Hz. With this bandwidth efficiency, a throughput of 6400 bps can be achieved on a telephone line having a bandwidth of 3200 Hz. The bandwidth efficiency of the MFM D16-QAM encoding format is four bits/s/Hz, since this format carries four bits per tone per baud. The D16-QAM signal can achieve a throughput of 12800 bps for this bandwidth efficiency. In comparison, the V.32 throughput is 9600 bps.

Though not shown in tabular form in this report, the bit error rate (BER) for DQPSK is lower than that of D16-QAM for every case of input SNR and baud length tested. The achievable throughput for DQPSK is half that of D16-QAM, but on a switched telephone network that generally has higher noise levels, such as overseas telephone lines, a trade-off of throughput for lower BER may be desirable. D16-QAM encoding is the best choice of the two formats tested for use

on telephone networks that have guaranteed maximum noise levels, such as the private telephone networks.

Areas of further study should include the design and implementation of a PC expansion board that contains both a transmitter and receiver to allow testing of full-duplex MFM communications on industry standard computers. Echo cancelling algorithms for MFM should be studied. Additionally, error control coding should be incorporated into the existing Pascal software for improvement of the BERs. Error control coding for MFM is the subject of a thesis by LT Robert W. Ives [Ref. 14]. Finally, a high-speed (V.32 compatible) modem utilizing an MFM encoding format should be designed and built for further testing of the MFM signal over a voice frequency channel.

APPENDIX A. MFM TRANSMITTER

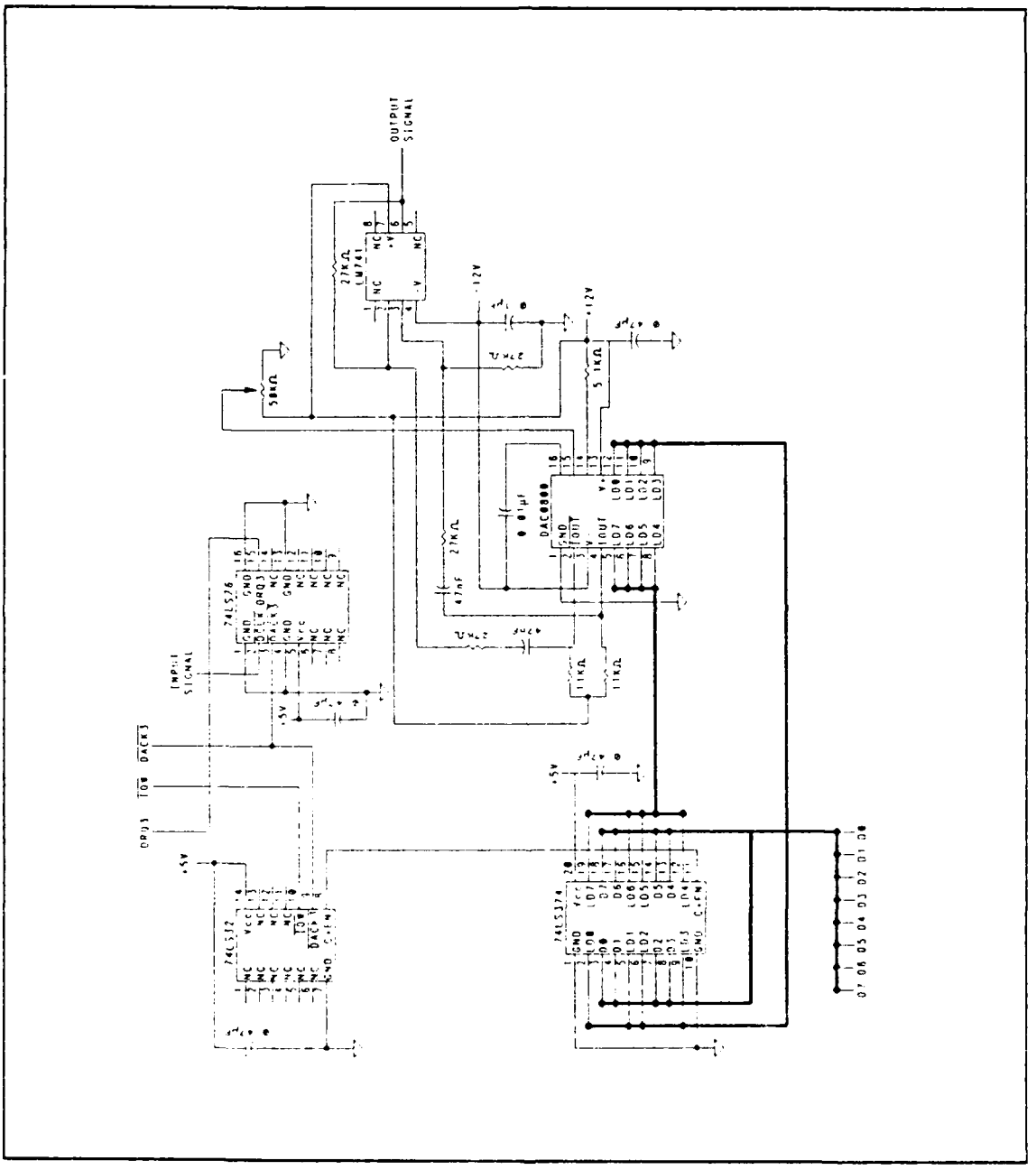


Figure 16. MFM transmitter expansion board schematic.

APPENDIX B. DQPSK TRANSMIT PROGRAM

```
program DQPSKXMIT;
(*Transmits a syncbaud and message from file 'MESSAGE.DAT'.
  The message is encoded using DQPSK. 'MESSAGE.DAT' is a text
  file. It should already exist before using this program.
  Output is used to collect data for TESTING*)

uses crt,graph,plrte55;

const
   FIRST_ELEMENT = -28929;

type
   TNvector = array[0..4095] of single;
   TNvectorPtr =  TNvector;
   BCSTARRAY = array[FIRST_ELEMENT..32767] of byte;

var
   kx,
   k1,k2,I,w,
   NUMBAUDS,MAXNUMBAUDS,
   BAUDCOUNT,BYTECOUNT,
   SYMBOLCOUNT,MAXNUMCHAR,
   MESSAGESIZE,dmachn,
   n2p,bk0psz,bk1psz,
   port,Aadd,proc            :integer;
   Badd                      :word;
   MAGNITUDE,
   CHARACTERS_PER_BAUD,
   xm,xp                     :single;
   XREAL,XIMAG               :TNvectorPtr;
   INVERSE                   :boolean;
   TEMPBYTE,ERROR            :byte;
   BCST                      :BCSTARRAY;
   BYTEFILE                  : file of byte;
   TESTFILE                  :text;
   ANSWER,
   NEXTCHAR                  :char;
   plbuf                     :array[0..768] of integer;


($L dmainit)
($L dmastop)

(*-----------------------------------------------------------*)

procedure Cnvttotime;
(*computes inverse FFT, returns values in XREAL *)

type
   pass     = array[0..8191] of single;
```

```
   passptr  =  pass;

var
   FVALUES   :passptr;

begin

     new(FVALUES);
     fillchar(FVALUES ,sizeof(FVALUES ),0);
     for i:= 0 to kx-1 do
         begin
              FVALUES [2*i]   := XREAL [i];
              FVALUES [2*i+1] := XIMAG [i];
         end;
     plxfto(FVALUES ,Aadd,2*kx);
     plwtxf;
     vfieee(Aadd,Aadd,2*kx);
     cifft(Aadd,n2p);
     cereal(Aadd,Badd,kx);
     vtieee(Badd,Badd,kx);
     plwtrn;
     plxffm(Badd,XREAL ,kx);
     plwtxf;
     dispose(FVALUES);
end;  (*Cnvttotime*)

(*-----------------------------------------------------------*)

procedure SyncBaud;
(*Process the synchronization baud and stores the 256 point
  time domain sequence at the beginning of the packet storage
  area.*)

var
   J, TEMP            :integer;
   SYNCDATA           :byte;
   SYNCMAG            :single;
   syncvals           :text;

begin
   assign(syncvals,'syncvals.dat');
   rewrite(syncvals);
   kx:=256;
   n2p:=8;
   SYNCMAG:= MAGNITUDE;
   fillchar(XREAL ,sizeof(XREAL ),0);
   fillchar(XIMAG ,sizeof(XIMAG ),0);
   XREAL [68]:= -SYNCMAG ; XIMAG [68]:= -SYNCMAG ;
   XREAL [69]:= -SYNCMAG ; XIMAG [69]:= -SYNCMAG ;
   XREAL [70]:= -SYNCMAG ; XIMAG [70]:=  SYNCMAG ;
   XREAL [71]:= -SYNCMAG ; XIMAG [71]:=  SYNCMAG ;
   XREAL [72]:=  SYNCMAG ; XIMAG [72]:= -SYNCMAG ;
   XREAL [73]:=  SYNCMAG ; XIMAG [73]:=  SYNCMAG ;
   XREAL [74]:= -SYNCMAG ; XIMAG [74]:=  SYNCMAG ;
   XREAL [75]:=  SYNCMAG ; XIMAG [75]:=  SYNCMAG ;
   XREAL [76]:= -SYNCMAG ; XIMAG [76]:=  SYNCMAG ;
```

```
   XREAL [77]:= -SYNCMAG ; XIMAG [77]:= -SYNCMAG ;
   XREAL [78]:=  SYNCMAG ; XIMAG [78]:= -SYNCMAG ;
   XREAL [79]:= -SYNCMAG ; XIMAG [79]:= -SYNCMAG ;
   XREAL [80]:=  SYNCMAG ; XIMAG [80]:=  SYNCMAG ;
   XREAL [81]:=  SYNCMAG ; XIMAG [81]:= -SYNCMAG ;
   XREAL [82]:= -SYNCMAG ; XIMAG [82]:= -SYNCMAG ;
   XREAL [83]:=  SYNCMAG ; XIMAG [83]:=  SYNCMAG ;

(*complex conjugate image*)
   for J := 68 to 83 do
      begin
         XREAL [256-J]:= XREAL [J];
         XIMAG [256-J]:=-XIMAG [J];
      end;  (*for J*)

Cnvttotime;  (*compute the 256 time domain values*)

     for J := 0 to 255 do              (*force values to range 0-255*)
      begin                            (*for d/a conversion*)
      if (XREAL [J] > 127) then
         begin
           writeln('syncvals exceed range of D/A converter');
           halt;
         end;
         TEMP:=round(XREAL [J] + 126);
         if TEMP < 0 then
            TEMP:=0;
         SYNCDATA:=TEMP;
         BCST[J+FIRST_ELEMENT]:=SYNCDATA;
       (*  writeln(syncvals,BCST[J+FIRST_ELEMENT]);  *)
      end;  (*for J*)
      close(syncvals);
end;  (*SyncBaud*)

(*-------------------------------------------------------------*)

procedure SelectBaud;
(*SelectBaud establishes kx, k1, and k2, and n2p*)

var
   ANSWER : integer;

begin
   kx:=0;
   repeat
      if kx < 0 then writeln('TRY AGAIN');
      writeln('What is the length of the bauds (kx)?');
      writeln('i.e. 256, 512, 1024, 2048, 4096');
      readln(ANSWER);
      case ANSWER of
      256: begin
             k1:=5; k2:=85; kx:=256; n2p:=8;
           end;
      512: begin
             k1:=10; k2:=170; kx:=512; n2p:=9;
           end;
```

```
      1024: begin
              k1:=20; k2:=340; kx:=1024; n2p:=10;
            end;
      2048: begin
              k1:=40; k2:=680; kx:=2048; n2p:=11;
            end;
      4096: begin
              k1:=80; k2:=1360; kx:=4096; n2p:=12;
            end;
      end; (*case kx*)
           if kx = 0 then kx := -1;
   until kx > 0;
end;  (*SelectBaud*)

(*-----------------------------------------------------------*)

procedure TailorPacket;
(*TailorPacket sets the maximum number of baud required to
  encode the message*)

begin
   MESSAGESIZE:= filesize(BYTEFILE);
   writeln('Messege is ',MESSAGESIZE,' bytes.');
   CHARACTERS_PER_BAUD:=(k2-k1)/4;
   MAXNUMCHAR:= trunc(10240.0/kx * CHARACTERS_PER_BAUD);
   if MESSAGESIZE > MAXNUMCHAR then
      begin
         writeln('Message is to large. The last ',
                  MESSAGESIZE - MAXNUMCHAR,
                 ' characters will not be transmitted.');
         MESSAGESIZE:=MAXNUMCHAR;
      end;
   MAXNUMBAUDS:=trunc(MESSAGESIZE / CHARACTERS_PER_BAUD);
   if frac(MESSAGESIZE / CHARACTERS_PER_BAUD) > 0.0 then
      MAXNUMBAUDS:=MAXNUMBAUDS + 1;
   repeat
    writeln;
    writeln('Enter number of ',kx,' bauds to process.  ',
             MAXNUMBAUDS,' is the maximum.');
    readln(NUMBAUDS);
   until NUMBAUDS in [1..MAXNUMBAUDS];
end;  (*TailorPacket*)

(*-----------------------------------------------------------*)

procedure DiffEncode;
(*DiffEncode differential encodes symbols on a tone-to-tone
  basis. BYTEFILE is read from one byte at a time. The byte
  is isolated into 2-bit groups and stored in BITS. BITS is
  then used to DQPSK encode the frequency domain arrays
  XREAL and XIMAG. Bytes partially encoded are carried over
  into the next baud by global variable TEMPBYTE.*)

var
   J         :integer;
   BITS      :byte;
```

```
begin
   fillchar(XREAL ,sizeof(XREAL ),0);
   fillchar(XIMAG ,sizeof(XIMAG ),0);

(*first tone of every baud set to pi/2*)
   XREAL [k1]:= MAGNITUDE;
   XIMAG [k1]:= MAGNITUDE;
   if SYMBOLCOUNT = 0 then
         read(BYTEFILE,TEMPBYTE);
   for J:= (k1 + 1) to k2 do
      begin
         SYMBOLCOUNT:=SYMBOLCOUNT + 1;
         if frac(SYMBOLCOUNT / 4) = 0.25 then
            BITS:= (TEMPBYTE and $C0) shr 6;
         if frac(SYMBOLCOUNT / 4) = 0.5 then
            BITS:= (TEMPBYTE and $30) shr 4;
         if frac(SYMBOLCOUNT / 4) = 0.75 then
            BITS:= (TEMPBYTE and $0C) shr 2;
         if frac(SYMBOLCOUNT / 4) = 0.0 then
            begin
               BITS:= TEMPBYTE and $03;
               if not EOF(BYTEFILE) then
                  read(BYTEFILE,TEMPBYTE)
               else
                  TEMPBYTE:=$40;  (*fill character*)
            end;
         if (BITS < 0) and (BITS > 3) then
            writeln('Bits not assigned properly');

(*differential encode*)
         case BITS of
            0: begin XREAL [J]:= XREAL [J-1];
                     XIMAG [J]:= XIMAG [J-1]; end;
            1: begin XREAL [J]:=-XIMAG [J-1];
                     XIMAG [J]:= XREAL [J-1]; end;
            2: begin XREAL [J]:= XIMAG [J-1];
                     XIMAG [J]:=-XREAL [J-1]; end;
            3: begin XREAL [J]:=-XREAL [J-1];
                     XIMAG [J]:=-XIMAG [J-1]; end;
         end;(*case BITS*)
      end;  (*for J*)

(*complex conjugate image*)
   for J:= k1 to k2 do
      begin
         XREAL [kx - J]:= XREAL [J];
         XIMAG [kx - J]:=-XIMAG [J];
         xp := arctan(ximag [j]/xreal [j]) * 180/pi;
         xm := sqrt(sqr(xreal [j])+sqr(ximag [j]));
         if (ximag [j] > 0) and (xreal [j] > 0) then
            xp := xp;
         if (ximag [j] > 0) and (xreal [j] < 0) then
            xp := (90+xp) + 90;
         if (ximag [j] < 0) and (xreal [j] < 0) then
            xp := 180 + xp;
```

```
            if (ximag [j] < 0) and (xreal [j] > 0) then
               xp := (90+xp) + 270;
         (*  if baudcount = 3 then  *)
              begin
                writeln(TESTFILE,baudcount,' ',J,'  ',xm,'  ',xp);
              end;
       end;
end;  (*DiffEncode*)

(*-----------------------------------------------------------*)

procedure ScaleData;
(*ScaleData converts each real value in XREAL down to a byte
  and stores the byte in the packet storage buffer, BCST.
  INDEX establishes the location in the buffer of each byte
  in the packet.*)

var
   INDEX,J,TEMP      :integer;
   DATA              :byte;

begin

   for J := 0 to kx-1 do
      begin
      if (XREAL [J] > 127) then
         begin
           writeln('broadcast values > 127',XREAL [J]:8:2);
       (*  halt;  *)
         end;
         TEMP := round(XREAL [J] + 126);
         if TEMP < 0 then
            TEMP := 0;
         DATA := TEMP;
      (*256 is added to INDEX to start message bauds
        after the sync baud*)
         INDEX := J+(BAUDCOUNT-1)*kx+FIRST_ELEMENT+256;
         BCST[INDEX] := DATA;
     (*  if baudcount = 3 then
         writeln(testfile,J:4,'  ',round(XREAL [J]));  *)
     end;  (*for J*)
    end;  (*ScaleData*)

(*-----------------------------------------------------------*)

 procedure Dmastop;external;
(*Masks DMA, stopping data transfer.*)

(*-----------------------------------------------------------*)

procedure Dmainit(var BCST:BCSTARRAY;BYTECOUNT:integer); external;
(*Assembly lanuage procedure used to initialize and unmask
  the DMA for data transfer. The source code must be
  converted to a BIN file.*)

(*-----------------------------------------------------------*)
```

```
begin
      dmachn:=0;
      plinit(dmachn,plbuf,sizeof(plbuf));
      plslib('C: PL1250 PLLIB.15');
      proc:=1;
      port:=$0318;
      bk0psz:=0;
      bk1psz:=1024;
      plsprc(proc,port,bk0psz,bk1psz);
      Aadd:=$0400;
      Badd:=$8400;

      new(XREAL);
      new(XIMAG);

(*contains hex values to be encoded and transmitted*)
      assign(BYTEFILE,'MESSAGE.DAT');
      reset(BYTEFILE);

(*Output file of encoded symbols. Used for system testing*)
      assign(TESTFILE,'XMITDAT.DAT');
      rewrite(TESTFILE);

      repeat
       writeln('Enter magnitude of tones.(greater than 65,
               less than 1501)');
       readln(MAGNITUDE);
      until MAGNITUDE > 0.0;

      writeln('Loading sync baud.');
      SyncBaud;
      SelectBaud;
      TailorPacket;

      SYMBOLCOUNT:=0;
      TEMPBYTE:=$00;
      writeln('Number of bauds is ',numbauds);

      for baudcount := 1 to numbauds do
         begin
            DiffEncode;
            writeln('Performing IFFT ',BAUDCOUNT,'    ',
                     NUMBAUDS-BAUDCOUNT,' left');
            Cnvttotime;
            ScaleData;
         end;  (*for BAUDCOUNT*)

      BYTECOUNT := 256 + NUMBAUDS*kx - 1;
      writeln(bytecount);

   repeat
      writeln('Press return to transmit');readln;
      Dmainit(BCST,BYTECOUNT);
      repeat
         writeln('Transmit some more? (*yes or no*) ');
```

```
            readln(ANSWER);
         until ANSWER in ['n','N','y','Y'];
         Dmastop;
      until ANSWER in ['n','N'];

         dispose(XREAL);
         dispose(XIMAG);
         close(BYTEFILE);
         close(TESTFILE);
   (*    reset(TESTFILE);
         while not EOF(TESTFILE) do
            begin
               while not EOLN(TESTFILE) do
                  begin
                     read(TESTFILE,NEXTCHAR);
                     write(NEXTCHAR);
                  end;
               readln(TESTFILE);
               writeln;
            end;
      close(TESTFILE);   *)
   end.
```

APPENDIX C. D16-QAM TRANSMIT PROGRAM

```
program DQAMXMIT;
(*Transmits a syncbaud and message from file 'MESSAGE.DAT'.
  The message is differentially encoded using 16-QAM. 'MESSAGE.DAT'
  is a text file. It should already exist before using this program.
  Output is used to collect data for TESTING*)

uses crt,plrte55;

const
   FIRST_ELEMENT = -28929;

type
   TNvector = array[0..4095] of single;
   TNvectorPtr =  TNvector;
   BCSTARRAY = array[FIRST_ELEMENT..32767] of byte;

var
   kx,
   k1,k2,I,w,
   NUMBAUDS,MAXNUMBAUDS,
   BAUDCOUNT,BYTECOUNT,
   SYMBOLCOUNT,MAXNUMCHAR,
   MESSAGESIZE,dmachn,
   n2p,bk0psz,bk1psz,
   port,Aadd,proc                    :integer;
   Badd                              :word;
   MAGNITUDE,
   CHARACTERS_PER_BAUD,
   PREV_TONE_MAGNITUDE,PREV_PHASE    :single;
   XREAL,XIMAG                       :TNvectorPtr;
   INVERSE                           :boolean;
   TEMPBYTE,ERROR                    :byte;
   BCST                              :BCSTARRAY;
   BYTEFILE                          : file of byte;
   TESTFILE                          :text;
   ANSWER,
   NEXTCHAR                          :char;
   plbuf                             :array[0..768] of integer;


($L dmainit)
($L dmastop)

(*-----------------------------------------------------------*)

procedure Cnvttotime;
(*computes inverse FFT, returns values in XREAL *)
type
   pass     =  array[0..8191] of single;
   passptr  =  pass;
```

```
var
   FVALUES   :passptr;

begin
     new(FVALUES);
     fillchar(FVALUES ,sizeof(FVALUES ),0);
     for i:= 0 to kx-1 do
         begin
              FVALUES [2*i]   := XREAL [i];
              FVALUES [2*i+1] := XIMAG [i];
         end;
     plxfto(FVALUES ,Aadd,2*kx);
     plwtxf;
     vfieee(Aadd,Aadd,2*kx);
     cifft(Aadd,n2p);
     cereal(Aadd,Badd,kx);
     vtieee(Badd,Badd,kx);
     plwtrn;
     plxffm(Badd,XREAL ,kx);
     plwtxf;
     dispose(FVALUES);
end;  (*Cnvttotime*)

(*-----------------------------------------------------------*)

procedure SyncBaud;
(*Process the synchronization baud and stores the 256 point
  time domain sequence at the beginning of the packet storage
  area.*)
var
   J, TEMP            : integer;
   SYNCDATA           : byte;
   SYNCMAG            : single;
   syncvals           :text;
begin
   assign(syncvals,'syncvals.dat');
   rewrite(syncvals);
   kx:=256;
   n2p:=8;
   SYNCMAG:= MAGNITUDE;
   fillchar(XREAL ,sizeof(XREAL ),0);
   fillchar(XIMAG ,sizeof(XIMAG ),0);
   XREAL [68]:= -SYNCMAG ; XIMAG [68]:= -SYNCMAG ;
   XREAL [69]:= -SYNCMAG ; XIMAG [69]:= -SYNCMAG ;
   XREAL [70]:= -SYNCMAG ; XIMAG [70]:=  SYNCMAG ;
   XREAL [71]:= -SYNCMAG ; XIMAG [71]:=  SYNCMAG ;
   XREAL [72]:=  SYNCMAG ; XIMAG [72]:= -SYNCMAG ;
   XREAL [73]:=  SYNCMAG ; XIMAG [73]:=  SYNCMAG ;
   XREAL [74]:= -SYNCMAG ; XIMAG [74]:=  SYNCMAG ;
   XREAL [75]:=  SYNCMAG ; XIMAG [75]:=  SYNCMAG ;
   XREAL [76]:= -SYNCMAG ; XIMAG [76]:=  SYNCMAG ;
   XREAL [77]:= -SYNCMAG ; XIMAG [77]:= -SYNCMAG ;
   XREAL [78]:=  SYNCMAG ; XIMAG [78]:= -SYNCMAG ;
   XREAL [79]:= -SYNCMAG ; XIMAG [79]:= -SYNCMAG ;
   XREAL [80]:=  SYNCMAG ; XIMAG [80]:=  SYNCMAG ;
```

```
   XREAL [81]:=  SYNCMAG ; XIMAG [81]:= -SYNCMAG ;
   XREAL [82]:= -SYNCMAG ; XIMAG [82]:= -SYNCMAG ;
   XREAL [83]:=  SYNCMAG ; XIMAG [83]:=  SYNCMAG ;

(*complex conjugate image*)
   for J := 68 to 83 do
      begin
         XREAL [256-J]:= XREAL [J];
         XIMAG [256-J]:=-XIMAG [J];
      end;  (*for J*)

Cnvttotime;  (*compute the 256 time domain values*)

     for J := 0 to 255 do            (*force values to range 0-255*)
      begin                          (*for d/a conversion*)
         TEMP:=round(XREAL [J] + 126);
         if TEMP < 0 then
            TEMP:=0;
         SYNCDATA:=TEMP;
         BCST[J+FIRST_ELEMENT]:=SYNCDATA;
       (*  writeln(syncvals,BCST[J+FIRST_ELEMENT]);  *)
      end;  (*for J*)
        close(syncvals);
end;  (*SyncBaud*)

(*-----------------------------------------------------------*)

procedure SelectBaud;
(*SelectBaud establishes kx, k1, and k2, and n2p*)

var
   ANSWER : integer;

begin
   kx:=0;
   repeat
      if kx < 0 then writeln('TRY AGAIN');
      writeln('What is the length of the bauds (kx)?');
      writeln('i.e. 256, 512, 1024, 2048, 4096');
      readln(ANSWER);
      case ANSWER of
      256: begin
             k1:=5; k2:=85; kx:=256; n2p:=8;
           end;
      512: begin
             k1:=10; k2:=170; kx:=512; n2p:=9;
           end;
     1024: begin
             k1:=20; k2:=340; kx:=1024; n2p:=10;
           end;
     2048: begin
             k1:=40; k2:=680; kx:=2048; n2p:=11;
           end;
     4096: begin
             k1:=80; k2:=1360; kx:=4096; n2p:=12;
           end;
```

```
      end;  (*case kx*)
           if kx = 0 then kx := -1;
   until kx > 0;
end;  (*SelectBaud*)

(*-----------------------------------------------------------*)

procedure TailorPacket;
(*TailorPacket sets the maximum number of baud required to
  encode the message*)

begin
   MESSAGESIZE:= filesize(BYTEFILE);
   writeln('Message is ',MESSAGESIZE,' bytes.');
   CHARACTERS_PER_BAUD:= (k2-k1)/2;      (*for qpsk:(k2-k1)/4;*)
   MAXNUMCHAR:= trunc(10240.0/kx * CHARACTERS_PER_BAUD);
   if MESSAGESIZE > MAXNUMCHAR then
      begin
         writeln('Message is to large. The last ',
                  MESSAGESIZE - MAXNUMCHAR,
                  ' characters will not be transmitted.');
         MESSAGESIZE:=MAXNUMCHAR;
      end;
   MAXNUMBAUDS:=trunc(MESSAGESIZE / CHARACTERS_PER_BAUD);
   if frac(MESSAGESIZE / CHARACTERS_PER_BAUD) > 0.0 then
      MAXNUMBAUDS:=MAXNUMBAUDS + 1;
   repeat
    writeln;
    writeln('Enter number of ',kx,' bauds to process.  ',
             MAXNUMBAUDS,' is the maximum.');
    readln(NUMBAUDS);
   until NUMBAUDS in [1..MAXNUMBAUDS];
end;  (*TailorPacket*)

(*-----------------------------------------------------------*)

procedure DiffEncode;
(*DiffEncode differentially encodes the message on a tone-to-tone
  basis. BYTEFILE is read from one byte at a time. The byte
  is isolated into two 4-bit groups. Then the first three bits
  in each symbol of 4 bits are used to determine the phase shift
  between tones, and the last bit of the 4 bit symbol is to
  determine the magnitude offset . The encoded tones are
  converted to rectangular coordinates and are stored in the arrays
  XREAL and XIMAG. Bytes partially encoded are carried over into the
  next baud by global variable TEMPBYTE *)

var
     SHORT_VECTOR,LONG_VECTOR,PHASESHIFT,
     TONE_MAGNITUDE,TONE_PHASE,
     PREV_TONE_PHASE,PREV_TONE_MAGNITUDE          :single;
     DELTAPHI,DELTAMAG                            :byte;
     J                                            :integer;

begin
     fillchar(XREAL ,sizeof(XREAL ),0);
```

```
      fillchar(XIMAG ,sizeof(XIMAG ),0);
      LONG_VECTOR := MAGNITUDE;
      SHORT_VECTOR := LONG_VECTOR*0.5;
      PREV_TONE_MAGNITUDE := SHORT_VECTOR;
      PREV_PHASE    := 22.5;

      XREAL [k1] := SHORT_VECTOR * cos(22.5*pi/180.0);
      XIMAG [k1] := SHORT_VECTOR * sin(22.5*pi/180.0);

   (*  if BAUDCOUNT = 3 then  *)
      writeln(TESTFILE,baudcount,' ',k1,'  ',PREV_TONE_MAGNITUDE,
              '  ',PREV_PHASE);

      if SYMBOLCOUNT = 0 then
           read(bytefile,TEMPBYTE);

      for J:= (k1 +1) to k2 do
        begin
          SYMBOLCOUNT := SYMBOLCOUNT + 1;

(*seperate magnitude/phase bits*)

          if frac(SYMBOLCOUNT/2) = 0.5 then
             begin
                  DELTAPHI := (TEMPBYTE and $E0) shr 5;
                  DELTAMAG := (TEMPBYTE and $10) shr 4;
             end;
          if frac(SYMBOLCOUNT/2) = 0.0 then
             begin
                  DELTAPHI := (TEMPBYTE and $0E) shr 1;
                  DELTAMAG := (TEMPBYTE and $01);
                  if NOT EOF(bytefile) then
                     read(bytefile,TEMPBYTE)
                  else
                     TEMPBYTE := $02;
             end;

(*differentially encode the last bit in the four bit symbol*)

          if  PREV_TONE_MAGNITUDE = SHORT_VECTOR then
              begin
                    case DELTAMAG of
                         0: TONE_MAGNITUDE := SHORT_VECTOR;
                         1: TONE_MAGNITUDE := LONG_VECTOR;
                    end;
              end  (*previous tone short case*)
           else  (*PREV_TONE_MAGNITUDE = LONG_VECTOR*)
              begin
                    case DELTAMAG of
                         0: TONE_MAGNITUDE  := LONG_VECTOR;
                         1: TONE_MAGNITUDE  := SHORT_VECTOR;
                    end;   (*end previous tone long case*)
              end;

(*Now use the first three bits in the symbol to determine the
  amount of phase rotation to the next encoded tone*)
```

```
        case DELTAPHI of
             0:PHASESHIFT := 0;
             1:PHASESHIFT := 45;
             2:PHASESHIFT := 135;
             3:PHASESHIFT := 90;
             4:PHASESHIFT := -45;
             5:PHASESHIFT := -90;
             6:PHASESHIFT := 180;
             7:PHASESHIFT := -135;
        end; (*case DELTAPHI of*)

(*Now assign the actual phase of the tone being encoded which is a
  function of the previous phase, and the phaseshift*)

        TONE_PHASE := PREV_PHASE + PHASESHIFT;
        if TONE_PHASE >= 360.0 then
           TONE_PHASE := TONE_PHASE - 360.0;

(*Now convert the magnitude and phase of the tone to rectangular
  coordinates*)

        XREAL [J] := TONE_MAGNITUDE * cos(TONE_PHASE*pi/180);
        XIMAG [J] := TONE_MAGNITUDE * sin(TONE_PHASE*pi/180);

(*Save the newly encoded tone's magnitude and phase for the next
   encoding iteration **)

        PREV_TONE_MAGNITUDE := TONE_MAGNITUDE;
        PREV_PHASE          := TONE_PHASE;

     (*  if BAUDCOUNT = 3 then  *)
        writeln(TESTFILE,baudcount,' ',J,'  ',TONE_MAGNITUDE,'  ',
                TONE_PHASE);

       end;  (*end of encoding proces for one tone,
               encode next tone*)

(*Put the complex conjugate of the encoded tones in the second half
  of the array before computing the IFFT for this baud*)

        for J:= k1 to k2 do
            begin
                 XREAL [kx - J]:= XREAL [J];
                 XIMAG [kx - J]:=-XIMAG [J];
             end;
end;  (*DiffEncode*)

(*-----------------------------------------------------------*)

procedure ScaleData;
(*ScaleData converts each real value in XREAL down to a byte
  and stores the byte in the packet storage buffer, BCST.
  INDEX establishes the location in the buffer of each byte
  in the packet.*)
```

```
var
   INDEX,J,TEMP      :integer;
   DATA              :byte;

begin
   for J := 0 to kx-1 do
      begin
      IF (xreal [J] > 127) then
         begin
           writeln('broadcast values > 127',xreal [J]:8:2);
       (*  halt;  *)
         end;
         TEMP := round(XREAL [J] + 126);
         if TEMP < 0 then
            TEMP := 0;
         DATA := TEMP;
      (*256 is added to INDEX to start message bauds
        after the sync baud*)
         INDEX := J+(BAUDCOUNT-1)*kx+FIRST_ELEMENT+256;
         BCST[ INDEX] := DATA;
       (*  if baudcount = 1 then
         writeln(testfile,J:4,'  ',round(XREAL [J]));  *)
     end;  (*for J*)
    end;  (*ScaleData*)

(*-----------------------------------------------------------*)

 procedure Dmastop;external;
(*Masks DMA, stopping data transfer.*)

(*-----------------------------------------------------------*)

procedure Dmainit(var BCST:BCSTARRAY;BYTECOUNT:integer)  ;external;
(*Assembly language procedure used to initialize and unmask
  the DMA for data transfer. The source code must be
  converted to a OBJ file.*)

(*-----------------------------------------------------------*)

begin
      dmachn:=0;
      plinit(dmachn,plbuf,sizeof(plbuf));
      plslib('C: PL1250 PLLIB.15');
      proc:=1;
      port:=$0318;
      bk0psz:=0;
      bk1psz:=1024;
      plsprc(proc,port,bk0psz,bk1psz);
      Aadd:=$0400;
      Badd:=$8400;

      new(XREAL);
      new(XIMAG);

(*contains hex values to be encoded and transmitted*)
      assign(BYTEFILE,'MESSGE.DAT');
```

```
      reset(BYTEFILE);

(*Output file of encoded symbols. Used for system testing*)
      assign(TESTFILE,'XMITDAT.DAT');
      rewrite(TESTFILE);

      repeat
       writeln('Enter magnitude of tones.(greater than 65, less than
               1501)');
       readln(MAGNITUDE);
      until MAGNITUDE > 0.0;

      writeln('Loading sync baud.');
      SyncBaud;
      SelectBaud;
      TailorPacket;

      SYMBOLCOUNT:=0;
      TEMPBYTE:=$00;
      writeln('Number of bauds is ',numbauds);

      for baudcount := 1 to numbauds do
         begin
            DiffEncode;
            writeln('Performing IFFT ',BAUDCOUNT,'    ',
                     NUMBAUDS-BAUDCOUNT,' left');
            Cnvttotime;
            ScaleData;
         end;  (*for BAUDCOUNT*)

      BYTECOUNT := 256 + NUMBAUDS*kx - 1;
      writeln(bytecount);

   repeat
      writeln('Press return to transmit');readln;
      Dmainit(BCST,BYTECOUNT);
      repeat
         writeln('Transmit some more? (*yes or no*) ');
         readln(ANSWER);
      until ANSWER in ['n','N','y','Y'];
      Dmastop;
   until ANSWER in ['n','N'];

      dispose(XREAL);
      dispose(XIMAG);
      close(BYTEFILE);
      close(TESTFILE);
end.
```

APPENDIX D. DQPSK RECEIVE PROGRAM

```
program DQPSKREC;
(*Acquires the signal. Stores it in a memory buffer.
  Differential decodes between tones. Maximum number of bauds
  are received. The number of bauds processed is a user input*)

uses Graph,Crt,tp55d16,plrte55;

($I-)
($R-)
const Max_Buffer = 65500;

type
(*TYPE for real and imaginary data for FFT routing*)
   TNvector = array[0..4095] of single;
   TNvectorPtr  =  TNvector;  (*Pointer for FFT data array
                               which allows dynamic
                               allocation of memory*)
   clr = array[0..4095] of word;
   clrptr = clr;

var
  INVERSE                      :boolean;
  XREAL,XIMAG                  :TNvectorPtr;
  color                        :clrptr;
  ERROR,TEMPBYTE               :byte;
  J,I,xradd,xroadd,proc,port,
  k1,k2,kx,ANSWER,ERR_CODE,
  BAUDCOUNT,SYMBOLCOUNT,n2p,
  NUMBAUDS,MAXNUMBAUDS,dmachn,
  bk0psz,bk1psz                :integer;
  MAGNITUDE,PHASE              :real;
  DATAVECTOR                   : integer;
  DMAPOINTER                   :pointer;
  OUTFILE,recdat               :TEXT;
  plbuf                        :array[0..4095] of integer;

(*-----------------------------------------------------------*)

procedure PacketSetUp;

begin
   repeat
      clrscr;
      if kx < 0 then writeln('TRY AGAIN');
      writeln('Enter baud size ');
      readln(ANSWER);
      case ANSWER of
         256:  begin
                    kx:= 256; n2p:=8; k1:=5; k2:=85;
               end;
```

```
            512:  begin
                       kx:= 512; n2p:=9; k1:=10; k2:=170;
                  end;
            1024: begin
                       kx:=1024; n2p:=10; k1:=20; k2:=340;
                  end;
            2048: begin
                       kx:=2048; n2p:=11; k1:=40; k2:=680;
                  end;
            4096: begin
                       kx:=4096; n2p:=12; k1:=80; k2:=1360;
                  end;
         end;  (*case*)

         if kx = 0 then kx := -1;
      until kx > 0;

      MAXNUMBAUDS := trunc((MAX_BUFFER/2)/kx);

      repeat
         writeln;
         writeln('Enter number of ',kx,' bauds to process. ',
                  MAXNUMBAUDS,' is the maximum.');
         readln(NUMBAUDS);
      until NUMBAUDS in [1..MAXNUMBAUDS];
   end;  (*PacketSetUp*)

   (*-----------------------------------------------------------*)

   procedure AcquireData;
   (*AcquireData initializes Metrobyte DASH-16F data acquisition
     board, using TTOOLS procedure D16_int and D16_ainm. Data
     transfer is controlled by the DMA controller and initialized
     by D16_ainm and disabled by D16_dma_int_disable. TTOOLS
     procedures are external procedures included by 'uses' tp4d16.*)

   var
       RATE:  real;
       I,CNT_NUM, MODE, CYCLE, TRIGGER,
       BASE_ADR, INT_LEVEL, DMA_LEVEL,
       BOARD_NUM, CHANLO,
       OP_TYPE, STATUS, NEXT_CNT, ERR_CODE_S :       integer;

    begin
      BOARD_NUM := 0; INT_LEVEL := 7;  DMA_LEVEL := 1;
      BASE_ADR := $300;

      D16_init(BOARD_NUM,BASE_ADR,INT_LEVEL,DMA_LEVEL,ERR_CODE);

      CHANLO := 0;
      CYCLE:=0;     (*0-one sweep of the DMA  1-autoinitialize*)
      TRIGGER:=0;   (*0 - external   1 - internal*)
      CNT_NUM:=32767; (*# of samples*)
      RATE := 10000.0;  (*used for internal trigger*)
      MODE := 2;     (*DMA mode*)
      writeln('Ready to acquire');
```

```
   D16_ainm(BOARD_NUM,CHANLO,MODE,CYCLE,TRIGGER,CNT_NUM,
            RATE, DATAVECTOR ,ERR_CODE);

   STATUS := 11;

 (*status indicates the progress of acquisition. When all
  samples have been acquired status=0*)
   repeat
      D16_dma_int_status(BOARD_NUM,OP_TYPE,STATUS,NEXT_CNT,
                         ERR_CODE_S);
   until STATUS = 0;
   writeln('Data received');

   if ERR_CODE <> 0 then
      D16_print_error(ERR_CODE);
   D16_dma_int_disable(BOARD_NUM,ERR_CODE);
end;  (*Acquire*)

(*-----------------------------------------------------------*)

procedure ConvertData;
(*ConvertData seperates channel and acquired data. CHAN_DATA
  is not used. Acquired data is stored in XREAL.*)

var
    AD_DATA:  array[0..4095] of integer;
    I,CHAN_DATA,ERR_CODE,
    SEGMENTPART,OFFSETPART      :integer;
    NEWDATAVECTOR               :integer;
    TEMPPOINTER                 :pointer;

begin
     fillchar(xreal ,sizeof(xreal ),0);
     fillchar(ximag ,sizeof(ximag ),0);
     SEGMENTPART:=seg(DATAVECTOR );
     OFFSETPART:=ofs(DATAVECTOR ) + 2 * kx * (BAUDCOUNT - 1);
     TEMPPOINTER:=ptr(SEGMENTPART,OFFSETPART);
     NEWDATAVECTOR := TEMPPOINTER;
     d16_convert_data(2047,kx,NEWDATAVECTOR ,AD_DATA[0],
                      CHAN_DATA,0,ERR_CODE);

     for I:= 0 to (kx - 1) do
         begin
              xreal [i]  := AD_data[i];
         end;
end;  (*ConvertData*)

(*-----------------------------------------------------------*)

procedure DiffDecode;
(*DiffDecode differentially decodes complex frequency domain
  arrays XREAL and XIMAG. Four decoded symbols are recombined
  into a byte and transferred to file BYTESOUT.DAT.*)

var
```

```
   I                    :integer;
   TEMPREAL,TEMPIMAG    :single;
   BITS                 :byte;
   TEMPCHAR             :char;

begin
   fillchar(color ,sizeof(color ),0);
   for I := k1 to (k2 - 1) do
      begin
           (*Complex multiply two adjacent tones, I and the complex
             conjugate of I+1. This will give the phase difference
             between the two tones. The answer is in rectangular
             notation*)
         TEMPREAL:=XREAL [I] * XREAL [I+1] +
                   XIMAG [I] * XIMAG [I+1];
         TEMPIMAG:=XREAL [I] * XIMAG [I+1] -
                   XREAL [I+1] * XIMAG [I];

            (*Complex multiply (TEMPREAL + j TEMPIMAGE) and (1+j).
             This rotates the differential vector pi/4 radians.
             XREAL [I] and XIMAG [I] are used to store the results.
             This eliminate the original data*)

         XREAL [I]:=(TEMPREAL - TEMPIMAG) / 80;
         XIMAG [I]:=(TEMPREAL + TEMPIMAG) / 80;

(*decode*)
         if (XREAL [I] >= 0) and (XIMAG [I] > 0) then
            begin
                 BITS:=$00; color [I] := 0;
            end;
         if (XREAL [I] < 0) and (XIMAG [I] > 0) then
            begin
                 BITS:=$01; color [I] := 10;
            end;
         if (XREAL [I] < 0) and (XIMAG [I] <= 0) then
            begin
                 BITS:=$03; color [I] := 14;
            end;
         if (XREAL [I] >= 0) and (XIMAG [I] <= 0) then
            begin
                 BITS:=$02; color [I] := 12;
            end;
            SYMBOLCOUNT := SYMBOLCOUNT + 1;

         (*fill TEMPBYTE with four symbols*)
         if frac(SYMBOLCOUNT / 4) = 0.25 then
            TEMPBYTE := (BITS shl 6);
         if frac(SYMBOLCOUNT / 4) = 0.5 then
            TEMPBYTE := (BITS shl 4) or TEMPBYTE;
         if frac(SYMBOLCOUNT / 4) = 0.75 then
            TEMPBYTE := (BITS shl 2) or TEMPBYTE;
         if (frac(SYMBOLCOUNT / 4) = 0.0) then
            begin
               TEMPBYTE := BITS or TEMPBYTE;
               TEMPCHAR := chr(TEMPBYTE);
```

```
                write(tempchar);
                TEMPBYTE:=0;
            end;  (*if frac*)
        end;  (*for I*)

        XREAL [k2]:=1;
        XIMAG [k2]:=1;
end;  (*DiffDecode*)

(*-----------------------------------------------------------*)

procedure viewphase;

var
   shade                                                :word;
   gd,gm,pta,ptb,pt1,pt2,ynzlo,ynzhi,yzlo,yzhi          :integer;

begin
     gd:=detect;
     initgraph(gd,gm,'C: TP DRIVERS');
     if graphresult <> grOk then
        halt(1);
     setgraphmode(1);
     setbkcolor(0);
     setcolor(15);

     (*draw axes*)
     line(50,0,50,140);
     line(50,140,590,140);
     line(50,180,50,320);
     line(50,320,590,320);

     (*compressed and zoom spectrum*)
     pta := 75;
     ptb := 555;
     yzlo := 140;
     yzhi := 15;
     pt1 := 50 + round(0.53 * (590-50));
     pt2 := 50 + round(0.70 * (590-50));
     ynzlo := 320;
     ynzhi := 320 - 130;
     line(pta,yzlo,pt1,ynzlo);
     line(pta+30*15,yzlo,pt1+48,ynzlo);
     i:= 68;
     repeat
           shade := color [i];
           setcolor(shade);
           line(pt1 + 3*(i-68),ynzlo,pt1 + 3*(i-68),ynzhi);
           if shade = 0 then
              begin
                   setfillstyle(ltslashfill,14);
                   bar(pta+30*(i-68)-3,yzlo,pta+30*(i-68)+3,yzhi);
              end
           else if shade > 0 then
              line(pta +30*(i-68),yzlo,pta +30*(i-68),yzhi);
           i := i+1;
```

```
      until(i=83);
      setcolor(14);
      settextjustify(centertext,centertext);
      outtextxy(300,10,'Zoom Spectrum');
      outtextxy(295,getmaxy div 2,'Compressed Spectrum');
      outtextxy(295,ynzlo +20,'Relative position of tone in 256 baud');
      outtextxy(pt1,ynzlo+10,'K1');
      outtextxy(pt1+16*3,ynzlo+10,'K2');
      outtextxy(590,ynzlo+10,'KX/2');
      repeat
       until(keypressed);
end;  (*viewphase*)

(*-----------------------------------------------------------*)

procedure Showmessage;
(*Showmessage read in decoded message*)

var
   NEXTCHAR: char;

begin
   writeln;
   writeln('The message transmitted is..');
   assign(OUTFILE,'MESSAGE.DAT');
   reset(OUTFILE);
   while not EOF(OUTFILE) do
      begin
         while not EOLN(OUTFILE) do
            begin
               read(OUTFILE,NEXTCHAR);
               write(NEXTCHAR);
            end;  (*while not EOLN*)
         readln(OUTFILE);
         writeln;
         end;  (*while not EOF*)
   close(OUTFILE);
end;  (*Showmessage*)

(*-----------------------------------------------------------*)

begin  (*main body*)

   dmachn:=0;
   plinit(dmachn,plbuf,sizeof(plbuf));
   plslib('C: PL850 PLLIB.15');
   proc:= 1;
   port:= $0318;
   bk0psz:=0;
   bk1psz:=1024;
   plsprc(proc,port,bk0psz,bk1psz);

   GetDMABuffer(MAX_BUFFER,DMAPOINTER,ERR_CODE);

   DATAVECTOR := DMAPOINTER; (*This statement assigns a
     generic pointer to a variable of a specific pointer
```

```
    type, i.e.  integer, so that the pointer can be
    passed to the d16_ainm routine.*)

    assign(recdat,'recdat.dat');
    rewrite(recdat);

  new(color);
  new(XREAL);
  new(XIMAG);

  ERROR := 0;
  kx:=0;
  SYMBOLCOUNT:=0;
  TEMPBYTE:=0;

  PacketSetUp;  (*determine baud lengths*)

  AcquireData;  (*AcquireData samples input analog signal*)

  xradd:=$0400;
  xrcadd:=$4400;

  for BAUDCOUNT := 1 to NUMBAUDS do
      begin
           ConvertData;
           plxfto(xreal ,xradd,kx);
           plwtxf;
           vfieee(xradd,xradd,kx);
           rfft(xradd,n2p);
           vtieee(xradd,xradd,kx);
           plwtrn;
           plxffm(xradd,xreal ,kx);
           plwtxf;

        for j:= 0 to kx div 2 do
           begin
                xreal [j] := xreal [2*j];
                ximag [j] := xreal [2*j+1];
           end;
        ximag [0] := 0;
        if baudcount = 3 then
           begin
           for i := k1 to k2 do
               begin
                    writeln(recdat,baudcount,' ',I,'  ',XREAL [I],
                            ' ',XIMAG [I]);
               end;
           end;
        DiffDecode;
      end;
      delay(1000);
      if kx = 256 then
         viewphase;

    close(recdat);
(*  close(OUTFILE);  *)
```

```
    dispose(XREAL);
    dispose(XIMAG);
    FreeDMABuffer(MAX_BUFFER,DMAPOINTER,ERR_CODE);
  (*  Showmessage;  *)
(*  writeln('Error = ',ERROR,'  hit the enter key');readln;  *)
end.
```

APPENDIX E. D16-QAM RECEIVE PROGRAM

```
program DQAMREC;
(*Acquires the signal. Stores it in a memory buffer.
  Differential decodes between tones. Maximum number of bauds
  are received. The number of bauds processed is a user input*)

uses Graph, Crt, tp55d16,plrte55;

($I-)
($R-)
const Max_Buffer = 65500;

type
(*TYPE for real and imaginary data for FFT routing*)
   TNvector = array[0..4095] of single;
   TNvectorPtr  =  TNvector;(*Pointer for FFT data array which
                      which allows dynamic allocation of memory*)

var
  INVERSE                          :boolean;
  XREAL,  XIMAG                    :TNvectorPtr;
  ERROR, TEMPBYTE                  :byte;
  J,I,xradd,xroadd,proc,port,
  k1,k2,kx,ANSWER,ERR_CODE,
  BAUDCOUNT,SYMBOLCOUNT,n2p,
  NUMBAUDS,MAXNUMBAUDS,dmachn,
  bk0psz,bk1psz                    :integer;
  MAGNITUDE,PHASE                  :real;
  DATAVECTOR                       : integer;
  DMAPOINTER                       :pointer;
  OUTFILE,recdat                   :TEXT;
  plbuf                            :array[0..4095] of integer;

(*-----------------------------------------------------------*)

procedure PacketSetUp;

begin
   repeat
      clrscr;
      if kx < 0 then writeln('TRY AGAIN');
      writeln('Enter baud size ');
      readln(ANSWER);
      case ANSWER of
         256:  begin
                    kx:= 256; n2p:=8; k1:=5; k2:=85;
               end;
         512:  begin
                    kx:= 512; n2p:=9; k1:=10; k2:=170;
               end;
         1024: begin
```

```
                    kx:=1024; n2p:=10; k1:=20; k2:=340;
                end;
            2048: begin
                    kx:=2048; n2p:=11; k1:=40; k2:=680;
                end;
            4096: begin
                    kx:=4096; n2p:=12; k1:=80; k2:=1360;
                end;
        end;  (*case*)

        if kx = 0 then kx := -1;
    until kx > 0;

    MAXNUMBAUDS := trunc((MAX_BUFFER/2)/kx);

    repeat
        writeln;
        writeln('Enter number of ',kx,' bauds to process. ',
                MAXNUMBAUDS,' is the maximum.');
        readln(NUMBAUDS);
    until NUMBAUDS in [1..MAXNUMBAUDS];

end;  (*PacketSetUp*)

(*-----------------------------------------------------------*)

procedure AcquireData;
(*AcquireData initializes Metrobyte DASH-16F data acquisition
  board, using TTOOLS procedure D16_int and D16_ainm. Data
  transfer is controlled by the DMA controller and initialized
  by D16_ainm and disabled by D16_dma_int_disable. TTOOLS
  procedures are external procedures included by 'uses' tp4d16.*)

var
    RATE:  real;
    I,CNT_NUM, MODE, CYCLE, TRIGGER,
    BASE_ADR, INT_LEVEL, DMA_LEVEL,
    BOARD_NUM, CHANLO,
    OP_TYPE, STATUS, NEXT_CNT, ERR_CODE_S        :integer;

 begin
    BOARD_NUM := 0; INT_LEVEL := 7;  DMA_LEVEL := 1;
    BASE_ADR := $300;

    D16_init(BOARD_NUM,BASE_ADR,INT_LEVEL,DMA_LEVEL,ERR_CODE);

    CHANLO := 0;
    CYCLE:=0;  (*0-one sweep of the DMA  1-autoinitialize*)
    TRIGGER:=0;  (*0 - external   1 - internal*)
    CNT_NUM:=32767;  (*# of samples*)
    RATE := 10000.0;  (*used for internal trigger*)
    MODE := 2;  (*DMA mode*)
    writeln('Ready to acquire');

    D16_ainm(BOARD_NUM,CHANLO,MODE,CYCLE,TRIGGER,CNT_NUM,
             RATE, DATAVECTOR ,ERR_CODE);
```

```
   STATUS := 11;
 (*status indicates the progress of acquisition. When all
   samples have been acquired status=0*)

   repeat
      D16_dma_int_status(BOARD_NUM,OP_TYPE,STATUS,NEXT_CNT,
                         ERR_CODE_S);
   until STATUS = 0;
   writeln('Data received');

   if ERR_CODE <> 0 then
      D16_print_error(ERR_CODE);
   D16_dma_int_disable(BOARD_NUM,ERR_CODE);
end;  (*Acquire*)

(*-----------------------------------------------------------*)

procedure ConvertData;
(*ConvertData seperates channel and acquired data. CHAN_DATA
  is not used. Acquired data is stored in XREAL.*)

var
    AD_DATA:  array[0..4095] of integer;
    I,CHAN_DATA,ERR_CODE,
    SEGMENTPART,OFFSETPART     :integer;
    NEWDATAVECTOR              :integer;
    TEMPPOINTER                :pointer;

begin
     fillchar(xreal ,sizeof(xreal ),0);
     fillchar(ximag ,sizeof(ximag ),0);
     SEGMENTPART:=seg(DATAVECTOR );
     OFFSETPART:=ofs(DATAVECTOR ) + 2 * kx * (BAUDCOUNT - 1);
     TEMPPOINTER:=ptr(SEGMENTPART,OFFSETPART);
     NEWDATAVECTOR := TEMPPOINTER;
     d16_convert_data(2047,kx,NEWDATAVECTOR ,AD_DATA[0],
                      CHAN_DATA,0,ERR_CODE);

     for I:= 0 to (kx - 1) do
         begin
              xreal [i]  := AD_data[i];
           (*  writeln(valsin,'Real in is ',xreal [i]:8:3,
                       ' at ',i:5);  *)
         end;
end;  (*ConvertData*)

(*-----------------------------------------------------------*)

procedure DiffDecode;
(*DiffDecode differentially decodes complex frequency domain
  arrays XREAL and XIMAG. Two decoded symbols are recombined
  into a byte and transferred to the screen*)

var
   I                                  :integer;
```

```
   TEMPREAL,TEMPIMAG,OLDMAG,NEWMAG  :single;
   BITS,PHASEBITS,MAGBIT            :byte;
   TEMPCHAR                         :char;

begin
   for I:= k1 to (k2-1) do
       begin
       SYMBOLCOUNT:= SYMBOLCOUNT + 1;

(*save the current and next magnitudes for future decoding*)

          OLDMAG := sqrt(sqr(XREAL [I]) + sqr(XIMAG [I]));
          NEWMAG := sqrt(sqr(XREAL [I+1]) + sqr(XIMAG [I+1]));

(*complex multiply adjacent tones to get phase differential*)

          TEMPREAL := XREAL [I] * XREAL [I+1] +
                      XIMAG [I] * XIMAG [I+1];
          TEMPIMAG := XREAL [I] * XIMAG [I+1] -
                      XREAL [I+1] * XIMAG [I] ;

(*now rotate phase by 22.5 degrees to line up with constellation
  phase sectors*)

                     XREAL [I] := 0.92 * TEMPREAL - 0.38 * TEMPIMAG;
                     XIMAG [I] := 0.92 * TEMPIMAG + 0.38 * TEMPREAL;

(*  writeln(freqs1,I,'  ',XREAL [I]:8:4,' ',XIMAG [I]:8:4);  *)
(*decode the phase difference into the first three bits of the
  symbol to be recovered*)

             PHASEBITS := $00;

          if (XREAL [I] > 0) and (XIMAG [I] > 0) then
             if XREAL [I] > XIMAG [I] then
                  PHASEBITS := $00
             else PHASEBITS := $02;

          if (XREAL [I] < 0) and (XIMAG [I] > 0) then
             if abs(XREAL [I]) > XIMAG [I] then
                  PHASEBITS := $04
             else PHASEBITS := $06;

          if (XREAL [I] < 0) and (XIMAG [I] < 0) then
             if abs(XREAL [I]) > abs(XIMAG [I]) then
                  PHASEBITS := $0C
             else PHASEBITS := $0E;

          if (XREAL [I] > 0) and (XIMAG [I] < 0) then
             if XREAL [I] > abs(XIMAG [I]) then
                  PHASEBITS := $08
             else PHASEBITS := $0A;

(*now differentially decode the magnitudes of the tones to get the
  fourth and last bit in the symbol*)
```

```
          if (NEWMAG > 1.5*OLDMAG) or (NEWMAG < 2*OLDMAG/3)
             then MAGBIT := $01
             else MAGBIT := $00;

(*now jam all the bits together*)

          (*fill TEMPBYTE with two symbols*)
          if frac(SYMBOLCOUNT / 2) = 0.5 then
             TEMPBYTE := ((PHASEBITS or MAGBIT) shl 4);
          if (frac(SYMBOLCOUNT / 2) = 0.0) then
             begin
               TEMPBYTE := (PHASEBITS or MAGBIT) or TEMPBYTE;
                TEMPCHAR := chr(TEMPBYTE);
                write(TEMPCHAR);  (*put ascii character to screen*)
              (*  write(OUTFILE,TEMPCHAR);  *)
                TEMPBYTE:=0;
             end;  (*if frac*)
      end;  (*for I*)
end;  (*DiffDecode*)

(*-----------------------------------------------------------*)

procedure Showmessage;  (*not used in this version*)
(*Showmessage read in decoded message*)

var
   NEXTCHAR: char;

begin
   writeln;
   writeln('The message transmitted is..');
   assign(OUTFILE,'MESSAGE.DAT');
   reset(OUTFILE);
   while not EOF(OUTFILE) do
      begin
         while not EOLN(OUTFILE) do
            begin
               read(OUTFILE,NEXTCHAR);
               write(NEXTCHAR);
            end;  (*while not EOLN*)
         readln(OUTFILE);
         writeln;
         end;  (*while not EOF*)
   close(OUTFILE);
end;  (*Showmessage*)

(*-----------------------------------------------------------*)

begin  (*main body*)

   dmachn:=0;
   plinit(dmachn,plbuf,sizeof(plbuf));
   plslib('c: pl1250 pllib.15');
   proc:= 1;
   port:= $0318;
   bk0psz:=0;
```

```
bk1psz:=1024;
plsprc(proc,port,bk0psz,bk1psz);

GetDMABuffer(MAX_BUFFER,DMAPOINTER,ERR_CODE);

DATAVECTOR := DMAPOINTER; (*This statement assigns a
  generic pointer to a variable of a specific pointer
  type, i.e.  integer, so that the pointer can be
  passed to the d16_ainm routine.*)

assign(recdat,'recdat.dat');
rewrite(recdat);

new(XREAL);
new(XIMAG);

ERROR := 0;
kx:=0;

PacketSetUp;

SYMBOLCOUNT:=0;
TEMPBYTE:=0;

AcquireData;  (*AcquireData samples input analog signal*)

xradd:=$0400;
xroadd:=$4400;

for BAUDCOUNT := 1 to NUMBAUDS do
    begin
          ConvertData;

          plxfto(xreal ,xradd,kx);
          plwtxf;
          vfieee(xradd,xradd,kx);
          rfft(xradd,n2p);
          vtieee(xradd,xradd,kx);
          plwtrn;
          plxffm(xradd,xreal ,kx);
          plwtxf;

      for j:= 0 to kx div 2 do
          begin
               xreal [j] := xreal [2*j];
               ximag [j] := xreal [2*j+1];
          end;
          ximag [0] := 0;

  (*  if baudcount = 3 then  *)
          for i := k1 to k2 do
             begin
                  writeln(recdat,baudcount,' ',I,' ',XREAL [I],
                  ' ',XIMAG [I]);
             end;
      DiffDecode;
```

```
        delay(500);
      end;
 (*  close(OUTFILE);  *)
   close(recdat);
   dispose(XREAL);
   dispose(XIMAG);
   FreeDMABuffer(MAX_BUFFER,DMAPOINTER,ERR_CODE);
(*  Showmessage;  *)
(*  writeln('Error = ',ERROR,'  hit the enter key');readln;
end.
```

APPENDIX F. SYNCHRONIZER PROGRAM

```
program SYNCLOAD;

uses crt;

type
    reference_array = array[1..128] of byte;

var
   j                    :integer;
   reference_values     :reference_array;
   num_ref_vals         :integer;
   vals                 :text;
   data                 :byte;
   testref2             :text;
($L I)

(*************************************************************************)
procedure I(var reference_values:reference_array;num_ref_vals:integer);
external;
(*************************************************************************)

begin
     assign(vals,'vals.dat');
     reset(vals);
     assign(testref2,'testref2.dat');
     reset(testref2);
     num_ref_vals := 127;
     for j := 1 to 128 do
     begin
         read(vals,data);
         reference_values[j] := data;
         writeln(reference_values[j]);
     end;
     I(reference_values,num_ref_vals);
     close(vals);
     close(testref2);
end.
```

APPENDIX G. DQPSK STATISTICS PROGRAM

```
program QPSKSNR;
(* This program uses the files XMITDAT.DAT and RECDAT.DAT to generate
a color plot of the errors in the received decoded tones. Green
indicates at least one phase decoding error in the ascii character.
(note that the file XMITDAT.DAT must be imported to the receive terminal
from the transmit terminal) *)

uses crt, graph;

var
   answer,answer2                          :char;
   i,j,n,rbaud,xbaud,rtone,xtone,
   baudcount,numbauds,k1,k2,kx,count,
   symbolcount,sector,b,btot,badbaud,
   numbits,badbaud2,bj,colorflag           :integer;
   xtempreal,rtempreal,xtempimag,
   rtempimag,totphaserrs,symerrs,del,
   sumr,sumi,tot,rmean,imean,varx,
   xmagr,xmagi,totsnr,snravg               :single;
   xbits,xphasebits,xmagbit,xtempbyte,
   rbits,rphasebits,rtempbyte,
   phasebitdiff,hue,pbd1,pbd2              :byte;
   xtempchar,rtempchar                     :char;
   xmitdat,recdat,output                   :text;
   xreal,ximag,rreal,rimag,xmag,xphase     :array[1..1280] of single;
   recdata                                 :array[1..48,1..120] of single;
   snrin                                   :string[4];

(-----------------------------------------------------------------)

begin   (main body)
   clrscr;
   assign(output,'output.dat');
   rewrite(output);
   assign(xmitdat,'xmitdat.dat');
   reset(xmitdat);
   assign(recdat,'recdat.dat');
   reset(recdat);
   writeln('Enter the input snr');
   readln(snrin);
   writeln('Enter the baud length ');
   readln(kx);
   writeln(output);
   writeln(output,'The baud length is ',kx,' and the SNRIN =', snrin);
   writeln('Enter the number of bauds to be processed');
   readln(numbauds);
   writeln('Throw out any bauds ? ');
   readln(answer);
   badbaud :=0;
   badbaud2:=0;
```

```
if answer in ['y','Y'] then
   begin
        writeln('Which baud ?');
        readln(badbaud);
        writeln('Any others ?');
        readln(answer2);
        if answer2 in ['y','Y'] then
           begin
           writeln('Enter baud #');
           readln(badbaud2);
           end;
   end;

case kx of
     256: begin
               k1:=5; k2:=85;
          end;
     512: begin
               k1:=10; k2:=170;
          end;
     1024: begin
               k1:=20; k2:=340;
          end;
     2048: begin
               k1:=40; k2:=680;
          end;
     4096: begin
               k1:=80; k2:=1360;
          end;
end; (case Kx)
TOTPHASERRS :=0;
SYMBOLCOUNT :=0;numbits:=0;
pbd1:=0; pbd2:=0; bj:=0;
totsnr:=0;
( count bit errors baud by baud )
(* read in transmit and receive values *)
for j:= 1 to numbauds do
    begin
         del:=0; rmean:=0; imean:=0;
         tot:=0; sumr:=0; sumi:=0;
         fillchar(recdata,sizeof(recdata),0);
         for i:= 1 to k2-k1+1 do
             begin
                  readln(xmitdat,xbaud,xtone,xmag[i],xphase[i]);
                  readln(recdat,rbaud,rtone,rreal[i],rimag[i]);

                  if (xbaud <> rbaud) or (xtone <> rtone) then
                     begin
                       writeln('RECDAT and XMITDAT do not match');
                       halt;
                     end;  (if xbaud)
                  xreal[i]:=xmag[i]*cos(xphase[i]*pi/180);
                  ximag[i]:=xmag[i]*sin(xphase[i]*pi/180);
             end;  (for read data files)
         writeln;
         write(j,' ');
```

```
            for I:= 1 to k2-k1 do
                begin
                    colorflag:=0;
                    symbolcount:=symbolcount+1;
(*complex multiply adjacent tones to get phase differential*)
                    XTEMPREAL := XREAL[ I] * XREAL[ I+1] +
                                 XIMAG[ I] * XIMAG[ I+1];
                    XTEMPIMAG := XREAL[ I] * XIMAG[ I+1] -
                                 XREAL[ I+1] * XIMAG[ I] ;
                    RTEMPREAL := RREAL[ I] * RREAL[ I+1] +
                                 RIMAG[ I] * RIMAG[ I+1];
                    RTEMPIMAG := RREAL[ I] * RIMAG[ I+1] -
                                 RREAL[ I+1] * RIMAG[ I];

(*now rotate phase by 45 degrees to line up with constellation
  phase sectors*)
                                 XREAL[ I] := (XTEMPREAL - XTEMPIMAG)/80;
                                 XIMAG[ I] := (XTEMPIMAG + XTEMPREAL)/80;
                    RREAL[ I] := (RTEMPREAL - RTEMPIMAG)/80;
                                 RIMAG[ I] := (RTEMPIMAG + RTEMPREAL)/80;

(*decode transmitted bits*)

                XBITS := $00;

            if (XREAL[ I] >= 0) and (XIMAG[ I] > 0) then
               begin
                    XBITS := $00;
               end;
            if (XREAL[ I] < 0) and (XIMAG[ I] > 0) then
               begin
                    XBITS := $01;
               end;
            if (XREAL[ I] < 0) and (XIMAG[ I] <= 0) then
               begin
                    XBITS := $03;
               end;
            if (XREAL[ I] >= 0) and (XIMAG[ I] <= 0) then
               begin
                    XBITS := $02;
               end;

(*decode the received bits*)

                    RBITS := $00;

            if (RREAL[ I] >= 0) and (RIMAG[ I] > 0) then
               begin
                    RBITS := $00;
               end;
            if (RREAL[ I] < 0) and (RIMAG[ I] > 0) then
               begin
                    RBITS := $01;
               end;
            if (RREAL[ I] < 0) and (RIMAG[ I] <= 0) then
               begin
```

```
                    RBITS := $03;
                end;
            if (RREAL[I] >= 0) and (RIMAG[I] <= 0) then
                begin
                    RBITS := $02;
                end;

(*determine the number of bit differences between the received decoded
  bits and the decoded transmitted bits*)

                    PHASEBITDIFF := XBITS xor RBITS;

                    if (j <> badbaud) and (j <> badbaud2) then
                    begin
                    case PHASEBITDIFF of
                    $01:  pbd1 :=pbd1+1;
                    $02:  pbd1 :=pbd1+1;
                    $03:  pbd2 :=pbd2+1;
                    end; (case PHASEBITDIFF)

 (*now count the total number of phase decoding errors*)

                    TOTPHASERRS := TOTPHASERRS + PHASEBITDIFF and $01;
                    TOTPHASERRS := TOTPHASERRS +
                                   (PHASEBITDIFF and $02) shr 1;
                    numbits:=numbits+2;
                  end;

(*assign colors to the text that is in error*)
                   if PHASEBITDIFF > 0 then
                      colorflag :=1;
 (*now jam all the bits together and color the errors*)

                   (*fill TEMPBYTE with four symbols*)
                   textcolor(15);
                   if frac(SYMBOLCOUNT / 4) = 0.25 then
                      begin
                      XTEMPBYTE := (XBITS shl 6);
                      RTEMPBYTE := (RBITS shl 6);
                      end;
                   if frac(SYMBOLCOUNT / 4) = 0.5 then
                      begin
                      XTEMPBYTE := (XBITS shl 4) or XTEMPBYTE;
                      RTEMPBYTE := (RBITS shl 4) or RTEMPBYTE;
                      end;
                   if frac(SYMBOLCOUNT / 4) = 0.75 then
                      begin
                      XTEMPBYTE := (XBITS shl 2) or XTEMPBYTE;
                      RTEMPBYTE := (RBITS shl 2) or RTEMPBYTE;
                      end;
                   if frac(SYMBOLCOUNT / 4) = 0.0 then
                      begin
                       if colorflag > 0 then
                       textcolor(138);    (1.green - phase error)
                        XTEMPBYTE := XBITS or XTEMPBYTE;
                        XTEMPCHAR := chr(XTEMPBYTE);
```

```
                        RTEMPBYTE := RBITS or RTEMPBYTE;
                        RTEMPCHAR := chr(RTEMPBYTE);
                        write(rtempchar);
                        textcolor(15);
                        XTEMPBYTE :=0;
                        RTEMPBYTE :=0;
                      end;  (*if frac*)
                end;  (*for I*)

(*now calculate the means and variances and snrout*)

            if (j <> badbaud) and (j <> badbaud2) then
            begin
            for I:=1 to k2-k1 do
              begin
                tot:=tot+1;
                sumr:=abs(RREAL[I])+sumr;
                sumi:=abs(RIMAG[I])+sumi;
              end;
              begin
                rmean:=sumr/tot;
                imean:=sumi/tot;
              end;
              begin
                del:=del+sqr(abs(RREAL[I])-rmean)+
                     sqr(abs(RIMAG[I])-imean);
              end;
            end;
            begin
              varx:=del/(2*tot);
              xmagr:=rmean/cos(45*pi/180.0);
              xmagi:=imean/sin(45*pi/180.0);
              snravg:=sqr((xmagr+xmagi)/2)/varx;
              totsnr:=totsnr+10*ln(snravg)/ln(10.0);
              bj:=bj+1;
            end;
        end;  (for j:= 1 to numbauds)
            writeln(output);
            writeln(output,'The overall SNROUT is  ',
                    (totsnr/bj):8:3,' db');
            writeln(output);
            writeln(output,'Total phase decoding bit errors = ',
                    TOTPHASERRS:5:0,
            ' out of ',numbits,' bits transmitted');
            writeln(output,'(',pbd1,
                    ' symbols with one bit phase decoding error)');
            writeln(output,'(',pbd2,
                    ' symbols with two bit phase decoding error)');
            close(recdat);
            close(xmitdat);
            close(output);
end.
```

APPENDIX H. D16-QAM STATISTICS PROGRAM

```
program QAMSNR;
(*This program uses the files XMITDAT.DAT and RECDAT.DAT to generate
a multi color plot of the errors in the received decoded tones. Yellow
indicates at least one magnitude decoding error in the ascii character,
green indicates at least one phase decoding error in the ascii
character and red indicates a combination of magnitude and phase
decoding errors in the ascii character. (note that the file XMITDAT.DAT
must be imported to the receive terminal from the transmit terminal)*)

uses crt, graph;

var
   answer,answer2                          :char;
   i,j,n,rbaud,xbaud,rtone,xtone,dtot,
   baudcount,numbauds,k1,k2,kx,count,
   symbolcount,sector,b,c,d,btot,ctot,
   badbaud,numbits,badbaud2,bj,cj,dj       :integer;
   xoldmag,roldmag,xtempreal,rtempreal,
   xnewmag,rnewmag,xtempimag,rtempimag,
   totphaserrs,totmagerrs,symerrs,
   smallmag,big,sml,del,meanbig,xmagbig,
   mbig,msml,obig,osml,mmeanbig,mmeansml,
   omeanbig,omeansml,mdel,odel,msnravg,
   osnravg,meansml,varx,snrbig,snrsml,
   xmagsml,snravg,mvarx,ovarx,msnrbig,
   msnrsml,osnrbig,osnrsml,mxmagbig,
   mxmagsml,oxmagbig,oxmagsml,totsnr,
   mtotsnr,ototsnr,bigmag                  :single;
   xbits,xphasebits,xmagbit,xtempbyte,
   rbits,rphasebits,rmagbit,rtempbyte,
   phasebitdiff,magbitdiff,hue,pbd1,
   pbd2,pbd3                               :byte;
   xtempchar,rtempchar                     :char;
   xmitdat,recdat,output                   :text;
   xreal,ximag,rreal,rimag,xmag,xphase     :array[1..1280] of single;
   statmat                                 :array[1..8,1..3] of single;
   recdata                                 :array[1..48,1..160] of single;
   snrin                                   :string[4];

(*----------------------------------------------------------------*)

procedure sort;

begin

     if (xmag[I]=smallmag) and (xmag[I+1]=smallmag) then
        begin
             statmat[sector,1]:=statmat[sector,1]+1;
             b := round(statmat[sector,1]);
             recdata[(2*sector)-1,b]:=RREAL[I];
```

```
            recdata[(2*sector),b]:=RIMAG[I];
        end
    else if (((xmag[I]=smallmag) and (xmag[I+1]=bigmag)) or
            ((xmag[I]=bigmag) and (xmag[I+1]=smallmag))) then
            begin
                statmat[sector,2]:=statmat[sector,2]+1;
                b := round(statmat[sector,2]);
                recdata[(2*sector)-1+16,b]:=RREAL[I];
                recdata[(2*sector)+16,b]:=RIMAG[I];
            end
    else  (*both xmag[I] and xmag[I+1] are large*)
        begin
            statmat[sector,3]:=statmat[sector,3]+1;
            b := round(statmat[sector,3]);
            recdata[(2*sector)-1+32,b]:=RREAL[I];
            recdata[(2*sector)+32,b]:=RIMAG[I];
        end;
end;

(*-----------------------------------------------------------------*)

begin  (*main body*)
   clrscr;
   assign(output,'output.dat');
   rewrite(output);
   assign(xmitdat,'xmitdat.dat');
   reset(xmitdat);
   assign(recdat,'recdat.dat');
   reset(recdat);
   writeln('Enter the input snr');
   readln(snrin);
   writeln('Enter the baud length ');
   readln(kx);
   writeln(output,'The baud length is ',kx,' and the SNRIN =',snrin);
   writeln('Enter the number of bauds to be processed');
   readln(numbauds);
   writeln('Enter the magnitude of the xmit short tones');
   readln(smallmag);
   bigmag := 2*smallmag;
   writeln('Throw out any bauds ? ');
   readln(answer);
   badbaud :=0;
   badbaud2:=0;
   if answer in ['y','Y'] then
      begin
           writeln('Which baud ?');
           readln(badbaud);
           writeln('Any others ?');
           readln(answer2);
           if answer2 in ['y','Y'] then
              begin
              writeln('Enter baud #');
              readln(badbaud2);
              end;
      end;
```

```
case kx of
     256: begin
               k1:=5; k2:=85;
          end;
     512: begin
               k1:=10; k2:=170;
          end;
     1024: begin
               k1:=20; k2:=340;
          end;
     2048: begin
               k1:=40; k2:=680;
          end;
     4096: begin
               k1:=80; k2:=1360;
          end;
end;  (*case Kx*)
TOTPHASERRS :=0;
TOTMAGERRS  :=0; SYMBOLCOUNT :=0; numbits:=0;
pbd1:=0; pbd2:=0; pbd3       :=0; bj:=0; cj:=0; dj:=0;
totsnr:=0; mtotsnr:=0; ototsnr:=0;

(*count bit errors baud by baud*)
(*read in transmit and receive values*)
for j:= 1 to numbauds do
    begin
         del:=0; big:=0; sml:=0; meanbig:=0; meansml:=0; btot:=0;
         mdel:=0; mbig:=0; msml:=0; mmeanbig:=0; mmeansml:=0; ctot:=0;
         odel:=0; obig:=0; osml:=0; omeanbig:=0; omeansml:=0; dtot:=0;
         fillchar(statmat,sizeof(statmat),0);
         fillchar(recdata,sizeof(recdata),0);
         for i:= 1 to k2-k1+1 do
             begin
                  readln(xmitdat,xbaud,xtone,xmag[i],xphase[i]);
                  readln(recdat,rbaud,rtone,rreal[i],rimag[i]);

                  if (xbaud <> rbaud) or (xtone <> rtone) then
                     begin
                          writeln('RECDAT and XMITDAT do not match');
                          halt;
                     end; (*if xbaud*)
                  (* convert the xmit vals to rectangular coordinates*)
                  xreal[i] := xmag[i]*cos(xphase[i]*pi/180);
                  ximag[i] := xmag[i]*sin(xphase[i]*pi/180);
             end;  (*for read data files*)
         writeln;
         write(j,' ');
         for I:= 1 to k2-k1 do
             begin
                 SYMBOLCOUNT:= SYMBOLCOUNT + 1;

(*save the current and next magnitudes for future decoding*)
                  XOLDMAG := sqrt(sqr(XREAL[I]) + sqr(XIMAG[I]));
                  XNEWMAG := sqrt(sqr(XREAL[I+1]) + sqr(XIMAG[I+1]));
                  ROLDMAG := sqrt(sqr(RREAL[I]) + sqr(RIMAG[I]));
                  RNEWMAG := sqrt(sqr(RREAL[I+1]) + sqr(RIMAG[I+1]));
```

```
(*complex multiply adjacent tones to get phase differential*)
                    XTEMPREAL := XREAL[ I] * XREAL[ I+1] +
                                 XIMAG[ I] * XIMAG[ I+1];
                    XTEMPIMAG := XREAL[ I] * XIMAG[ I+1] -
                                 XREAL[ I+1] * XIMAG[ I] ;
                    RTEMPREAL := RREAL[ I] * RREAL[ I+1] +
                                 RIMAG[ I] * RIMAG[ I+1];
                    RTEMPIMAG := RREAL[ I] * RIMAG[ I+1] -
                                 RREAL[ I+1] * RIMAG[ I];

(*now rotate phase by 22.5 degrees to line up with constellation
  phase sectors*)
                       XREAL[ I] := 0.92 * XTEMPREAL - 0.38 * XTEMPIMAG;
                       XIMAG[ I] := 0.92 * XTEMPIMAG + 0.38 * XTEMPREAL;
            RREAL[ I] := 0.92 * RTEMPREAL - 0.38 * RTEMPIMAG;
                       RIMAG[ I] := 0.92 * RTEMPIMAG + 0.38 * RTEMPREAL;

(*decode the transmit phase difference into the first three bits of the
  symbol to be recovered*)
                    XPHASEBITS := $00;

                    if (XREAL[ I] > 0) and (XIMAG[ I] > 0) then
                       if XREAL[ I] > XIMAG[ I] then
                          begin
                               XPHASEBITS := $00;
                               sector :=1;
                               sort;
                          end
                       else
                          begin
                               XPHASEBITS := $02;
                               sector :=2;
                               sort;
                          end;

                    if (XREAL[ I] < 0) and (XIMAG[ I] > 0) then
                       if abs(XREAL[ I] ) > XIMAG[ I] then
                          begin
                               XPHASEBITS := $04;
                               sector := 4;
                               sort;
                          end
                        else
                          begin
                               XPHASEBITS := $06;
                               sector := 3;
                               sort;
                          end;

                    if (XREAL[ I] < 0) and (XIMAG[ I] < 0) then
                       if abs(XREAL[ I] ) > abs(XIMAG[ I] ) then
                         begin
                               XPHASEBITS := $0C;
                               sector := 5;
                               sort;
```

```
                    end
                  else
                    begin
                          XPHASEBITS := $0E;
                          sector:=6;
                          sort;
                    end;
                if (XREAL[I] > 0) and (XIMAG[I] < 0) then
                  if XREAL[I] > abs(XIMAG[I]) then
                    begin
                          XPHASEBITS := $08;
                          sector := 8;
                          sort;
                    end
                  else
                    begin
                          XPHASEBITS := $0A;
                          sector := 7;
                          sort;
                    end;

(*decode the received phase difference into the first three bits of the
  symbol to be recovered*)
                RPHASEBITS := $00;

                if (RREAL[I] > 0) and (RIMAG[I] > 0) then
                  if RREAL[I] > RIMAG[I] then
                    RPHASEBITS := $00
                  else RPHASEBITS := $02;

                if (RREAL[I] < 0) and (RIMAG[I] > 0) then
                  if abs(RREAL[I]) > RIMAG[I] then
                    RPHASEBITS := $04
                  else RPHASEBITS := $06;

                if (RREAL[I] < 0) and (RIMAG[I] < 0) then
                  if abs(RREAL[I]) > abs(RIMAG[I]) then
                    RPHASEBITS := $0C
                  else RPHASEBITS := $0E;

                if (RREAL[I] > 0) and (RIMAG[I] < 0) then
                  if RREAL[I] > abs(RIMAG[I]) then
                    RPHASEBITS := $08
                  else RPHASEBITS := $0A;

(*determine the number of bit differences between the received decoded
  phasebits and the decoded transmitted phasebits*)

                 PHASEBITDIFF := XPHASEBITS xor RPHASEBITS;
                 if (j <> badbaud) and (j <> badbaud2) then
                 begin
                 case PHASEBITDIFF of
                 $01:  pbd1 :=pbd1+1;
                 $02:  pbd1 :=pbd1+1;
                 $04:  pbd1 :=pbd1+1;
                 $08:  pbd1 :=pbd1+1;
```

```
                $03:  pbd2 :=pbd2+1;
                $05:  pbd2 :=pbd2+1;
                $06:  pbd2 :=pbd2+1;
                $09:  pbd2 :=pbd2+1;
                $0A:  pbd2 :=pbd2+1;
                $0C:  pbd2 :=pbd2+1;
                $07:  pbd3 :=pbd3+1;
                $0B:  pbd3 :=pbd3+1;
                $0D:  pbd3 :=pbd3+1;
                $0E:  pbd3 :=pbd3+1;
                end;  (*case PHASEBITDIFF*)

(*now count the total number of phase decoding errors*)

          TOTPHASERRS := TOTPHASERRS + PHASEBITDIFF and $01;
          TOTPHASERRS := TOTPHASERRS + (PHASEBITDIFF and $02) shr 1;
          TOTPHASERRS := TOTPHASERRS + (PHASEBITDIFF and $04) shr 2;
          TOTPHASERRS := TOTPHASERRS + (PHASEBITDIFF and $08) shr 3;
          end;

(*now differentially decode the magnitudes of the tones to get the
  fourth and last bit in the symbol*)

          if (XNEWMAG > 1.5*XOLDMAG) or (XNEWMAG < 2*XOLDMAG/3)
             then XMAGBIT := $01
             else XMAGBIT := $00;
          if (RNEWMAG > 1.5*ROLDMAG) or (RNEWMAG < 2*ROLDMAG/3)
             then RMAGBIT := $01
             else RMAGBIT := $00;
          if (j <> badbaud) and (j <> badbaud2) then
             begin
               TOTMAGERRS := TOTMAGERRS + (XMAGBIT xor RMAGBIT);
               numbits:=numbits+4;
             end;
(*assign colors to the text that is in error*)
              if PHASEBITDIFF > 0 then
                  textcolor(138);    (*1.green - phase error*)
              if RMAGBIT <> XMAGBIT then
                  textcolor(142);    (*yellow  - mag error*)
              if (RMAGBIT <> XMAGBIT) and (PHASEBITDIFF <> 0) then
                  textcolor(140);    (*1. red  - dual error*)
              if (RMAGBIT = XMAGBIT) and (PHASEBITDIFF = 0) then
                  textcolor(15);

(*now jam all the bits together and color the errors*)

        (*fill TEMPBYTE with two symbols*)
        if frac(SYMBOLCOUNT / 2) = 0.5 then
              begin
                  hue := textattr;
                  XTEMPBYTE := ((XPHASEBITS or XMAGBIT) shl 4);
                  RTEMPBYTE := ((RPHASEBITS or RMAGBIT) shl 4);
              end;(* if frac *)
        if (frac(SYMBOLCOUNT / 2) = 0.0) then
          begin
            if (hue = 140) or (textattr = 140) then textcolor(140);
```

```
                if (hue = 142) and (textattr = 138) then textcolor(140);
                if (hue = 138) and (textattr = 142) then textcolor(140);
                if (hue = 142) and (textattr = 15) then textcolor(142);
                if (hue = 138) and (textattr = 15) then textcolor(138);
                      XTEMPBYTE := (XPHASEBITS or XMAGBIT) or XTEMPBYTE;
                      XTEMPCHAR := chr(XTEMPBYTE);
                      RTEMPBYTE := (RPHASEBITS or RMAGBIT) or RTEMPBYTE;
                if (RTEMPBYTE = $20) and (textattr <> 15) then
                      RTEMPBYTE := $5f;
                      RTEMPCHAR := chr(RTEMPBYTE);
                      (*put ascii character to screen*)
                       write(rtempchar);
                       XTEMPBYTE:=0;
                       RTEMPBYTE:=0;
              end;  (*if frac*)
            end;  (*for decode xmit and rec data*)

(*now calculate the means and variances and snrout*)

             if (j <> badbaud) and (j <> badbaud2) then
             begin
              for i:= 1 to 8 do
                begin
                     b:=round(statmat[i,1]);
                     btot:=btot+b;
                     c:=round(statmat[i,2]);
                     ctot:=ctot+c;
                     d:=round(statmat[i,3]);
                     dtot:=dtot+d;
                     if (i=1) or (i=4) or (i=5) or (i=8) then
                       begin
                         for count := 1 to b do
                           begin
                              big:=abs(recdata[(2*i)-1,count])+big;
                              sml:=abs(recdata[(2*i),count])+sml;
                           end;
                         for count := 1 to c do
                           begin
                              mbig:=abs(recdata[(2*i)-1+16,count])+mbig;
                              msml:=abs(recdata[(2*i)+16,count])+msml;
                           end;
                         for count := 1 to d do
                           begin
                              obig:=abs(recdata[(2*i)-1+32,count])+obig;
                              osml:=abs(recdata[(2*i)+32,count])+osml;
                           end;
                       end
                     else
                       begin
                         for count := 1 to b do
                           begin
                              big:=abs(recdata[(2*i),count])+big;
                              sml:=abs(recdata[(2*i)-1,count])+sml;
                           end;
                         for count := 1 to c do
                           begin
```

```
                    mbig:=abs(recdata[(2*i)+16,count])+mbig;
                    msml:=abs(recdata[(2*i)-1+16,count])+msml;
                  end;
                for count := 1 to d do
                  begin
                    obig:=abs(recdata[(2*i)+32,count])+obig;
                    osml:=abs(recdata[(2*i)-1+32,count])+osml;
                  end;
              end;
          end;

if btot > 1 then
begin
meanbig := big/btot;
meansml := sml/btot;
end;
if ctot > 1 then
begin
mmeanbig := mbig/ctot;
mmeansml := msml/ctot;
end;
if dtot > 1 then
begin
omeanbig := obig/dtot;
omeansml := osml/dtot;
end;

for i:= 1 to 8 do
  begin
        b:=round(statmat[i,1]);
        c:=round(statmat[i,2]);
        d:=round(statmat[i,3]);
        if (i=1) or (i=4) or (i=5) or (i=8) then
           begin
                for count := 1 to b do
                    begin
           del:=del+sqr(abs(recdata[(2*i)-1,count])-meanbig)+
                    sqr(abs(recdata[(2*i),count])-meansml);
                    end;
                for count := 1 to c do
                    begin
           mdel:=mdel+sqr(abs(recdata[(2*i)-1+16,count])-
                    mmeanbig)+sqr(abs(recdata[(2*i)+16,
                    count])-mmeansml);
                    end;
                for count := 1 to d do
                    begin
           odel:=odel+sqr(abs(recdata[(2*i)-1+32,count])-
                    omeanbig)+sqr(abs(recdata[(2*i)+32,
                    count])-omeansml);
                    end;
           end
        else
           begin
                for count := 1 to b do
                    begin
```

```
                    del:=del+sqr(abs(recdata[(2*i),count])-meanbig)+
                              sqr(abs(recdata[(2*i)-1,count])-meansml);
                              end;
                         for count := 1 to c do
                              begin
                    mdel:=mdel+sqr(abs(recdata[(2*i)+16,count])-
                              mmeanbig)+sqr(abs(recdata[(2*i)-1+16,
                              count])-mmeansml);
                              end;
                         for count := 1 to d do
                              begin
                    odel:=odel+sqr(abs(recdata[(2*i)+32,count])-
                              omeanbig)+sqr(abs(recdata[(2*i)-1+32,
                              count])-omeansml);
                              end;
                    end;
          end;

     if (btot > 1) then
     begin
     varx:=del/(2*btot);
     snrbig:=sqr(meanbig)/varx;
     snrsml:=sqr(meansml)/varx;
     xmagbig:=meanbig/cos(22.5*pi/180.0);
     xmagsml:=meansml/sin(22.5*pi/180.0);
     snravg:=sqr((xmagbig+xmagsml)/2)/varx;
     totsnr:=totsnr+10*ln(snravg)/ln(10.0);
     bj:=bj+1;
     end;

     if (ctot>1) then
     begin
     mvarx:=mdel/(2*ctot);
     msnrbig:=sqr(mmeanbig)/mvarx;
     msnrsml:=sqr(mmeansml)/mvarx;
     mxmagbig:=mmeanbig/cos(22.5*pi/180.0);
     mxmagsml:=mmeansml/sin(22.5*pi/180.0);
     msnravg:=sqr((mxmagbig+mxmagsml)/2)/mvarx;
     mtotsnr:=mtotsnr+10*ln(msnravg)/ln(10.0);
     cj:=cj+1;
     end;

     if (dtot >1) then
     begin
     ovarx:=odel/(2*dtot);
     osnrbig:=sqr(omeanbig)/ovarx;
     osnrsml:=sqr(omeansml)/ovarx;
     oxmagbig:=omeanbig/cos(22.5*pi/180.0);
     oxmagsml:=omeansml/sin(22.5*pi/180.0);
     osnravg:=sqr((oxmagbig+oxmagsml)/2)/ovarx;
     ototsnr:=ototsnr+10*ln(osnravg)/ln(10.0);
     dj:=dj+1;
     end;

(*   writeln(output,'The SNROUT for baud ',j,' is ',
     10*ln(snrbig)/ln(10.0):8:3,'db for the inner big means');
```

```
        writeln(output,'The SNROUT for baud ',j,' is ',
        10*ln(snrsml)/ln(10.0):8:3,'db for the inner small means');
        writeln(output,'The SNROUT for baud ',j,' is ',
        10*ln(snravg)/ln(10.0):8:3,'db for the inner averaged means');
        writeln(output,'The SNROUT for baud ',j,' is ',
        10*ln(msnrbig)/ln(10.0):8:3,'db for the middle big means');
        writeln(output,'The SNROUT for baud ',j,' is ',
        10*ln(msnrsml)/ln(10.0):8:3,'db for the middle small means');
        writeln(output,'The SNROUT for baud ',j,' is ',
        10*ln(msnravg)/ln(10.0):8:3,'db for the middle averaged means');
        writeln(output,'The SNROUT for baud ',j,' is ',
        10*ln(osnrbig)/ln(10.0):8:3,'db for the outer big means');
        writeln(output,'The SNROUT for baud ',j,' is  ',
        10*ln(osnrsml)/ln(10.0):8:3,'db for the outer small means');
        writeln(output,'The SNROUT for baud ',j,' is ',
        10*ln(osnravg)/ln(10.0):8:3,'db for the outer averaged means');
        writeln(output);  *)
        end;
    end;  (*for j := 1 to numbauds*)
    writeln(output);
    writeln(output,'The overall inner SNROUT is  ',(totsnr/bj):8:3,
            ' db');
    writeln(output,'The overall middle SNROUT is ',(mtotsnr/cj):8:3,
                   ' db');
    writeln(output,'The overall outer SNROUT is  ',(ototsnr/dj):8:3,
                   ' db');
    writeln(output);
    writeln(output,'Total phase decoding bit errors = ',
                   TOTPHASERRS:5:0);
    writeln(output,'(',pbd1,' symbols with one bit phase decoding
                   error)');
    writeln(output,'(',pbd2,' symbols with two bits phase decoding
                    error)');
    writeln(output,'(',pbd3,' symbols with three bits phase decoding
                   error)');
    writeln(output,'Total magnitude decoding bit errors = ',
                   TOTMAGERRS:5:0);
    writeln(output,'out of ',numbits,' bits transmitted');
    close(recdat);
    close(xmitdat);
    close(output);
end.
```

LIST OF REFERENCES

1. Paul H. Moose, "Theory of multi-frequency modulation (MFM) digital communications," Technical Report No. NPS-62-89-019, Naval Postgraduate School, Monterey, California, May 1989

2. Terry K. Gantenbein, "Implementation of multi-frequency modulation on an industry standard computer," Master's Thesis, Naval Postgraduate School, Monterey, California, September 1989

3. Peter G. Basil, "Real time multi-frequency modulation using differentially encoded signal constellations," Master's Thesis, Naval Postgraduate School, Monterey, California, June 1990

4. The International Telegraph and Telephone Consultative Committee (CCITT) Red Book vol. V, 8th Plenary Assembly, Malaga-Torremolinos, 1984

5. Johna Till, "Black magic: building a V.32 modem," *Electronic Design* , Vol. 37, No. 5, pp. 47-57, 1989

6. K. Sam Shanmugam, *Digital and Analog Communication Systems,* John Wiley & Sons, Inc., New York, 1979

7. John A.C. Bingham, *The Theory and Practice of Modem Design,* John Wiley & Sons, Inc., New York, 1988

8. William D. Glass, "DSP quashes echoes in V.32 modems," *Electronic Design*, Vol. 37, No. 8, pp.137-142, 1989

9. Jack Douglas, "V.32 modems are breaking the echo barrier," *Data Communications*, Vol. 17, No. 4, pp. 187-194, 1988

10. Leon W. Couch, *Digital and Analog Communication Systems,* Macmillan Publishing Co., New York, 1987

11. Robert Daniel Childs, "High speed output interface for a multifrequency quaternary phase shift keying signal generated on an industry standard computer," Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988

12. National Semiconductor Corp., *Linear Databook*, 1982

13. Paul H. Moose, "A progress report on communications digital signal processing: theory and performance of frequency domain differentially encoded

multi-frequency modulation," Technical Report No. NPS-62-90-012, Naval Postgraduate School, Monterey, California, June 1990

14. Robert W. Ives, "Error control coding for multi-frequency modulation," Master's Thesis, Naval Postgraduate School, Monterey, California, June 1990

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Professor P.H. Moose, Code EC/Me Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	6
5.	Professor J.H. Miller, Code EC/Mr Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
6.	Commander Naval Ocean Systems Center Attn: Mr. Darrell Marsh (Code 624) San Diego, CA 92151	3
7.	Commanding Officer USS Juneau (LPD-10) Attn: Lcdr Charles P. Salsman FPO San Francisco, CA 96669-1713	3